

Freedesktop Planet - Latest News

- [Sebastian Wick: How Hard Is It To Open a File?](#) (2026/04/23 20:41)

It's a question I had to ask myself multiple times over the last few months. Depending on the context the answer can be: very simple, just call the standard library function extremely hard, don't trust anything. If you are an app developer, you're lucky and it's almost always the first answer. If you develop something with a security boundary which involves files in any way, the correct answer is very likely the second one. Opening a File, the Hard Way Like so often, the details depend on the specifics, but in the worst-case scenario, there is a process on either side of the security boundary, which operate on a filesystem tree which is shared by both processes. Let's say that the process with more privileges operates on a file on behalf of the process with less privileges. You might want to restrict this to files in a certain directory, to prevent the less privileged process from, for example, stealing your SSH key, and thus take a subpath that is relative to that directory. The first obvious problem is that the subpath can refer to files outside of the directory if it contains ... If the privileged process gets called with a subpath of `../ssh/id_ed25519`, you are in trouble. Easy fix: normalize the path, and if we ever go outside of the directory, fail. The next issue is that every component of the path might be a symlink. If the privileged process gets called with a subpath of `link`, and `link` is a symlink to `../ssh/id_ed25519`, you might be in trouble. If the process with less privileges cannot create files in that part of the tree, it cannot create a malicious symlink, and everything is fine. In all other scenarios, nothing is fine. Easy fix: resolve the symlinks, expand the path, then normalize it. This is usually where most people think we're done, opening a file is not that hard after all, we can all do more fun things now. Really, this is where the fun begins. The fix above works, as long as the less privileged process cannot change the file system tree anywhere in the file's path while the more privileged process tries to access it. Usually this is the case if you unpack an attacker-provided archive into a directory the attacker does not have access to. If it can however, we have a classic TOCTOU (time-of-check to time-of-use) race. We have the path `foo/id_ed25519`, we resolve the symlinks, we expand the path, we normalize it, and while we did all of that, the other process just replaced the regular directory `foo` that we just checked with a symlink which points to `../ssh`. We just checked that the path resolves to a path inside the target directory though, and happily open the path `foo/id_ed25519` which now points to your ssh key. Not an easy fix. So, what is the fundamental issue here? A path string like `/home/user/.local/share/flatpak/app/org.example.App/deploy` describes a location in a filesystem namespace. It is not a reference to a file. By the time you finish speaking the path aloud, the thing it names may have changed. The safe primitive is the file descriptor. Once you have an fd pointing at an inode, the kernel pins that inode. The directory can be unlinked, renamed, or replaced with a symlink; the fd does not care. A common misconception is that file descriptors represent open files. It is true that they can do that, but fds opened with `O_PATH` do not require opening the file, but still provide a stable reference to an inode. The lesson that should be learned here is that you should not call any privileged process with a path. Period. Passing in file descriptors also has the benefit that they serve as proof that the calling process actually has access to the resource. Another important lesson is that dropping down from a file descriptor to a path makes everything racy again. For example, let's say that we want to bind mount something based on a file descriptor, and we only have the traditional mount API, so we convert the fd to a path, and pass that to mount. Unfortunately for the user, the kernel resolves the symlinks in the path that an attacker might have managed to place there. Sometimes it's possible to detect the issue after the fact, for example by checking that the inode and device of the mounted file and the file

descriptor match. With that being said, sometimes it is not entirely avoidable to use paths, so let's also look into that as well! In the scenario above, we have a directory in which we want all the paths to resolve in, and that the attacker does not control. We can thus open it with `O_PATH` and get a file descriptor for it without the attacker being able to redirect it somewhere else. With the `openat` syscall, we can open a path relative to the fd we just opened. It has all the same issues we discussed above, except that we can also pass `O_NOFOLLOW`. With that flag set, if the last segment of the path is a symlink, it does not follow it and instead opens the actual symlink inode. All the other components can still be symlinks, and they still will be followed. We can however just split up the path, and open the next file descriptor for the next path segment and resolve symlinks manually until we have done so for the entire path. `libglnx` `chase` `libglnx` is a utility library for GNOME C projects that provides fd-based filesystem operations as its primary API. Functions like `glnx_openat_ronly`, `glnx_file_replace_contents_at`, and `glnx_tmpfile_link_at` all take directory fds and operate relative to them. The library is built around the discipline of "always have an fd, never use an absolute path when you can use an fd." The most recent addition is `glnx_chaseat`, which provides safe path traversal, and was inspired by `systemd`'s `chase()`, and does precisely what was described above. `int glnx_chaseat (int dirfd, const char *path, GlnxChaseFlags flags, GError **error);` It returns an `O_PATH | O_CLOEXEC` fd for the resolved path, or `-1` on error. The real magic is in the flags: `typedef enum _GlnxChaseFlags { /* Default */ GLNX_CHASE_DEFAULT = 0, /* Disable triggering of automounts */ GLNX_CHASE_NO_AUTOMOUNT = 1 << 1, /* Do not follow the path's right-most component. When the path's right-most * component refers to symlink, return O_PATH fd of the symlink. */ GLNX_CHASE_NOFOLLOW = 1 << 2, /* Do not permit the path resolution to succeed if any component of the * resolution is not a descendant of the directory indicated by dirfd. */ GLNX_CHASE_RESOLVE_BENEATH = 1 << 3, /* Symlinks are resolved relative to the given dirfd instead of root. */ GLNX_CHASE_RESOLVE_IN_ROOT = 1 << 4, /* Fail if any symlink is encountered. */ GLNX_CHASE_RESOLVE_NO_SYMLINKS = 1 << 5, /* Fail if the path's right-most component is not a regular file */ GLNX_CHASE_MUST_BE_REGULAR = 1 << 6, /* Fail if the path's right-most component is not a directory */ GLNX_CHASE_MUST_BE_DIRECTORY = 1 << 7, /* Fail if the path's right-most component is not a socket */ GLNX_CHASE_MUST_BE_SOCKET = 1 << 8, } GlnxChaseFlags;` While it doesn't sound too complicated to implement, a lot of details are quite hairy. The implementation uses `openat2`, `open_tree` and `openat` depending on what is available and what behavior was requested, it handles auto-mount behavior, ensures that previously visited paths have not changed, and a few other things. An Aside on Standard Libraries The POSIX APIs are not great at dealing with the issue. The GLib/Gio APIs (`GFile`, etc.) are even worse and only accept paths. Granted, they also serve as a cross-platform abstraction where file descriptors are not a universal concept. Unfortunately, Rust also has this cross-platform abstraction which is based entirely on paths. If you use any of those APIs, you very likely created a vulnerability. The deeper issue is that those path-based APIs are often the standard way to interact with files. This makes it impossible to reason about the security of composed code. You can audit your own code meticulously, open everything with `O_PATH | O_NOFOLLOW`, chain `*at()` calls carefully — and then call a third-party library that calls `open(path)` internally. The security property you established in your code does not compose through that library call. This means that any system-level code that cares about filesystem security has to audit all transitive dependencies or avoid them in the first place. So what would a better GLib cross-platform API look like? I would say not too different from `chaseat()`, but returning opaque handles instead of file descriptors, which on Unix would carry the `O_PATH` file descriptor and a path that can be used for printing, debugging and things like that. You would open files from those handles, which would yield another kind of opaque handle for reading, writing, and so on. The current `GFile` was also designed to implement GVfs: `g_file_new_for_uri("smb://server/share/file")` gives you a `GFile` you can `g_file_read()` just like a local file. This is the right goal, but the wrong abstraction layer. Instead, this kind of access should be provided by FUSE, and the URI should be translated to a path on a specific

FUSE mount. This would provide a few benefits: The fd-chasing approach works everywhere because it is a real filesystem managed by the kernel. The filesystem becomes independent of GLib and can be used for example from Rust as well. It stacks with other FUSE filesystems, such as the XDG Desktop Document Portal used by Flatpak. Wait, Why Are You Talking About This? Nowadays I maintain a small project called Flatpak. Codean Labs recently did a security analysis on it and found a number of issues. Even though Flatpak developers were aware of the dangers of filesystems, and created libglx because of it, most of the discovered issues were just about that. One of them (CVE-2026-34078) was a complete sandbox escape. flatpak run was designed as a command-line tool for trusted users. When you type flatpak run org.example.App, you control the arguments. The code that processes the arguments was written assuming the caller is legitimate. It accepted path strings, because that's what command-line tools accept. The Flatpak portal was then built as a D-Bus service that sandboxed apps could call to start subsandboxes — and it did this by effectively constructing a flatpak run invocation and executing it. This connected a component designed for trusted input directly to an untrusted caller (the sandboxed app). Once that connection exists, every assumption baked into flatpak run about caller trustworthiness becomes a potential vulnerability. The fix wasn't "change one function" — it was "audit the entire call chain from portal request to bubblewrap execution and replace every path string with an fd." That's commits touching the portal, flatpak-run, flatpak_run_app, flatpak_run_setup_base_argv, and the bwrap argument construction, plus new options (--app-fd, --usr-fd, --bind-fd, --ro-bind-fd) threaded through all of them. If the GLib standard file and path APIs were secure, we would not have had this issue. Another annoyance here is that the entire subsandboxing approach in Flatpak comes from 15 years ago, when unprivileged user namespaces were not common. Nowadays we could (and should) let apps use kernel-native unprivileged user namespaces to create their own subsandboxes. Unfortunately with rather large changes comes a high likelihood of something going wrong. For a few days we scrambled to fix a few regressions that prevented Steam, WebKit, and Chromium-based apps from launching. Huge thanks to Simon McVittie! In the end, we managed to fix everything, made Flatpak more secure, the ecosystem is now better equipped to handle this class of issues, and hopefully you learned something as well.

- [Matthias Klumpp: Hello old new "Projects" directory!](#) (2026/04/18 08:06)

If you have recently installed a very up-to-date Linux distribution with a desktop environment, or upgraded your system on a rolling-release distribution, you might have noticed that your home directory has a new folder: "Projects" Why? With the recent 0.20 release of xdg-user-dirs we enabled the "Projects" directory by default. Support for this has already existed since 2007, but was never formally enabled. This closes a more than 11 year old bug report that asked for this feature. The purpose of the Projects directory is to give applications a default location to place project files that do not cleanly belong into one of the existing categories (Documents, Music, Pictures, Videos). Examples of this are software engineering projects, scientific projects, 3D printing projects, CAD design or even things like video editing projects, where project files would end up in the "Projects" directory, with output video being more at home in "Videos". By enabling this by default, and subsequently in the coming months adding support to GLib, Flatpak, desktops and applications that want to make use of it, we hope to give applications that do operate in a "project-centric" manner with mixed media a better default storage location. As of now, those tools either default to the home directory, or will clutter the "Documents" folder, both of which is not ideal. It also gives users a default organization structure, hopefully leading to less clutter overall and better storage layouts. This sucks, I don't like it! As usual, you are in control and can modify your system's behavior. If you do not like the "Projects" folder, simply delete it! The xdg-user-dirs utility will not try to create it again, and instead adjust the default location for this directory to your home directory. If you want more control, you can influence exactly what goes where by editing your ~/.config/user-dirs.dirs configuration file. If you are a system administrator or distribution vendor and want to set default locations for the default XDG directories, you

can edit the `/etc/xdg/user-dirs.defaults` file to set global defaults that affect all users on the system (users can still adjust the settings however they like though). What else is new? Besides this change, the 0.20 release of `xdg-user-dirs` brings full support for the Meson build system (dropping Automake), translation updates, and some robustness improvements to its code. We also fixed the “arbitrary code execution from unsanitized input” bug that the Arch Linux Wiki mentions here for the `xdg-user-dirs` utility, by replacing the shell script with a C binary. Thanks to everyone who contributed to this release!

- [Peter Hutterer: Huion devices in the desktop stack](#) (2026/04/14 05:49)

This post attempts to explain how Huion tablet devices currently integrate into the desktop stack. I'll touch a bit on the Huion driver and the OpenTablet driver but primarily this explains the intended integration[1]. While I have access to some Huion devices and have seen reports from others, there are likely devices that are slightly different. Huion's vendor ID is also used by other devices (UCLogic and Gaomon) so this applies to those devices as well. This post was written without AI support, so any errors are organic artisan hand-crafted ones. Enjoy. The graphics tablet stack First, a short overview of the ideal graphics tablet stack in current desktops. At the bottom is the physical device which contains a significant amount of firmware. That device provides something resembling the HID protocol over the wire (or bluetooth) to the kernel. The kernel typically handles this via the generic HID drivers [2] and provides us with an `/dev/input/event` evdev node, ideally one for the pen (and any other tool) and one for the pad (the buttons/rings/wheels/dials on the physical tablet). `libinput` then interprets the data from these event nodes, passes them on to the compositor which then passes them via Wayland to the client. Here's a simplified illustration of this: Unlike the X11 api, `libinput`'s API works both per-tablet and per-tool basis. In other words, when you plug in a tablet you get a `libinput` device that has a tablet tool capability and (optionally) a tablet pad capability. But the tool will only show up once you bring it into proximity. Wacom tools have sufficient identifiers that we can a) know what tool it is and b) get a unique serial number for that particular device. This means you can, if you wanted to, track your physical tool as it is used on multiple devices. No-one [3] does this but it's possible. More interesting is that because of this you can also configure the tools individually, different pressure curves, etc. This was possible with the `xf86-input-wacom` driver in X but only with some extra configuration, `libinput` provides/requires this as the default behaviour. The most prominent case for this is the eraser which is present on virtually all pen-like tools though some will have an eraser at the tail end and others (the numerically vast majority) will have it hardcoded on one of the buttons. Changing to eraser mode will create a new tool (the eraser) and bring it into proximity - that eraser tool is logically separate from the pen tool and can thus be configured differently. [4] Another effect of this per-tool behaviour is also that we know exactly what a tool can do. If you use two different styli with different capabilities (e.g. one with tilt and 2 buttons, one without tilt and 3 buttons), they will have the right bits set. This requires `libwacom` - a library that tells us, simply: any tool with id `0x1234` has `N` buttons and capabilities `A`, `B` and `C`. `libwacom` is just a bunch of static text files with a C library wrapped around those. Without `libwacom`, we cannot know what any individual tool can do - the firmware and kernel always expose the capability set of all tools that can be used on any particular tablet. For example: wacom's devices support an airbrush tool so any tablet plugged in will announce the capabilities for an airbrush even though >99% of users will never use an airbrush [5]. The compositor then takes the `libinput` events, modifies them (e.g. pressure curve handling is done by the compositor) and passes them via the Wayland protocol to the client. That protocol is a pretty close mirror of the `libinput` API so it works mostly the same. From then on, the rest is up to the application/toolkit. Notably, `libinput` is a hardware abstraction layer and conversion of hardware events into others is generally left to the compositor. IOW if you want a button to generate a key event, that's done either in the compositor or in the application/toolkit. But the current versions of `libinput` and the Wayland protocol do support all hardware features we're currently aware of: the various stylus types (including

Wacom's lens cursor and mouse-like "puck" devices) and buttons, rings, wheels/dials, and touchstrips on pads. We even support the rather once-off Dell Canvas Totem device. Huion devices Huion's devices are HID compatible which means they "work" out of the box but they come in two different modes, let's call them firmware mode and tablet mode. Each tablet device pretends to be three HID devices on the wire and depending on the mode some of those devices won't send events. Firmware mode This is the default mode after plugging the device in. Two of the HID devices exposed look like a tablet stylus and a keyboard. The tablet stylus is usually correct (enough) to work OOTB with the generic kernel drivers, it exports the buttons, pressure, tilt, etc. The buttons and strips/wheels/dials on the tablet are configured to send key events. For example, the Inspiroy 2S I have sends b/i/e/Ctrl+S/space/Ctrl+Alt+z for the buttons and the roller wheel sends Ctrl-/Ctrl= depending on direction. The latter are often interpreted as zoom in/out so hooray, things work OOTB. Other Huion devices have similar bindings, there is quite some overlap but not all devices have exactly the same key assignments for each button. It does of course get a lot more interesting when you want a button to do something different - you need to remap the key event (ideally without messing up your key map lest you need to type an 'e' later). The userspace part is effectively the same, so here's a simplified illustration of what happens in kernel land: Any vendor-specific data is discarded by the kernel (but in this mode that HID device doesn't send events anyway). Tablet mode If you read a special USB string descriptor from the English language ID, the device switches into tablet mode. Once in tablet mode, the HID tablet stylus and keyboard devices will stop sending events and instead all events from the device are sent via the third HID device which consists of a single vendor-specific report descriptor (read: 11 bytes of "here be magic"). Those bits represent the various features on the device, including the stylus features and all pad features as buttons/wheels/rings/strips (and not key events!). This mode is the one we want to handle the tablet properly. The kernel's hid-uclogic driver switches into tablet mode for supported devices, in userspace you can use e.g. huion-switcher. The device cannot be switched back to firmware mode but will return to firmware mode once unplugged. Once we have the device in tablet mode, we can get true tablet data and pass it on through our intended desktop stack. Alas, like ogres there are layers. hid-uclogic and udev-hid-bpf Historically and thanks in large parts to the now-discontinued digimend project, the hid-uclogic kernel driver did do the switching into tablet mode, followed by report descriptor mangling (inside the kernel) so that the resulting devices can be handled by the generic HID drivers. The more modern approach we are pushing for is to use udev-hid-bpf which is quite a bit easier to develop for. But both do effectively the same thing: they overlay the vendor-specific data with a normal HID report descriptor so that the incoming data can be handled by the generic HID kernel drivers. This will look like this: Notable here: the stylus and keyboard may still exist and get event nodes but never send events[6] but the uclogic/bpf-enabled device will be proper stylus/pad event nodes that can be handled by libinput (and thus the rest), with raw hardware data where buttons are buttons. Challenges Because in true manager speak we don't have problems, just challenges. And oh boy, we collect challenges as if we'd be organising the olympics. hid-uclogic and libinput First and probably most embarrassing is that hid-uclogic has a different way of exposing event nodes than what libinput expects. This is largely my fault for having focused on Wacom devices and internalized their behaviour for long years. The hid-uclogic driver exports the wheels and strips on separate event nodes - libinput doesn't handle this correctly (or at all). That'd be fixable but the composers also don't really expect this so there's a bit more work involved but the immediate effect is that those wheels/strips will likely be ignored and not work correctly. Buttons and pens work. udev-hid-bpf and huion-switcher hid-uclogic being a kernel driver has access to the underlying USB device. The HID-BPF hooks in the kernel currently do not, so we cannot switch the device into tablet mode from a BPF, we need it in tablet mode already. This means a userspace tool (read: huion-switcher) triggered via udev on plug-in and before the udev-hid-bpf udev rules trigger. Not a problem but it's one more moving piece that needs to be present (but boy, does this feel like the unix way...). Huion's precious product IDs By far the most annoying

part about anything Huion is that until relatively recently (I don't have a date but maybe until 2 years ago) all of Huion's devices shared the same few USB product IDs. For most of these devices we worked around it by matching on device names but there were devices that had the same product id and device name. At some point libwacom and the kernel and huion-switcher had to implement firmware ID extraction and matching so we could differ between devices with the same 0256:006d usb IDs. Luckily this seems to be in the past now with modern devices now getting new PIDs for each individual device. But if you have an older device, expect difficulties and, worse, things to potentially break after firmware updates when/if the firmware identification string changes. udev-hid-bpf (and uclogic) rely on the firmware strings to identify the device correctly. edit: and of course less than 24h after posting this I process a bug report about two completely different new devices sharing one of the product IDs udev-hid-bpf and hid-uclogic Because we have a changeover from the hid-uclogic kernel driver to the udev-hid-bpf files there are rough edges on "where does this device go". The general rule is now: if it's not a shared product ID (see above) it should go into udev-hid-bpf and not the uclogic driver. Easier to maintain, much more fire-and-forget. Devices already supported by udev-hid-bpf will remain there, we won't implement BPFs for those (older) devices, doubly so because of the aforementioned libinput difficulties with some hid-uclogic features. Reverse engineering required The newer tablets are always slightly different so we basically need to reverse-engineer each tablet to get it working. That's common enough for any device but we do rely on volunteers to do this. Mind you, the udev-hid-bpf approach is much simpler than doing it in the kernel, much of it is now copy-paste and I've even had quite some success to get e.g. Claude Code to spit out a 90% correct BPF on its first try. At least the advantage of our approach to change the report descriptor means once it's done it's done forever, there is no maintenance required because it's a static array of bytes that doesn't ever change. Plumbing support into userspace Because we're abstracting the hardware, userspace needs to be fully plumbed. This was a problem last year for example when we (slowly) got support for relative wheels into libinput, then wayland, then the compositors, then the toolkits to make it available to the applications (of which I think none so far use the wheels). Depending on how fast your distribution moves, this may mean that support is months and years off even when everything has been implemented. On the plus side these new features tend to only appear once every few years. Nonetheless, it's not hard to see why the "just sent Ctrl=, that'll do" approach is preferred by many users over "probably everything will work in 2027, I'm sure". So, what stylus is this? A currently unsolved problem is the lack of tool IDs on all Huion tools. We cannot know if the tool used is the two-button + eraser PW600L or the three-button-one-is-an-eraser-button PW600S or the two-button PW550 (I don't know if it's really 2 buttons or 1 button + eraser button). We always had this problem with e.g. the now quite old Wacom Bamboo devices but those pens all had the same functionality so it just didn't matter. It would matter less if the various pens would only work on the device they ship with but it's apparently quite possible to use a 3 button pen on a tablet that shipped with a 2 button pen OOTB. This is not difficult to solve (pretend to support all possible buttons on all tools) but it's frustrating because it removes a bunch of UI niceties that we've had for years - such as the pen settings only showing buttons that actually existed. Anyway, a problem currently in the "how I wish there was time" basket. Summary Overall, we are in an ok state but not as good as we are for Wacom devices. The lack of tool IDs is the only thing not fixable without Huion changing the hardware[7]. The delay between a new device release and driver support is really just dependent on one motivated person reverse-engineering it (our BPFs can work across kernel versions and you can literally download them from a successful CI pipeline). The hid-uclogic split should become less painful over time and the same as the devices with shared USB product IDs age into landfill and even more so if libinput gains support for the separate event nodes for wheels/strips/... (there is currently no plan and I'm somewhat questioning whether anyone really cares). But other than that our main feature gap is really the ability for much more flexible configuration of buttons/wheels/... in all compositors - having that would likely make the requirement for OpenTabletDriver and the Huion tablet

disappear. OpenTabletDriver and Huion's own driver The final topic here: what about the existing non-kernel drivers? Both of these are userspace HID input drivers which all use the same approach: read from a `/dev/hidraw` node, create a `uinput` device and pass events back. On the plus side this means you can do literally anything that the input subsystem supports, at the cost of a context switch for every input event. Again, a diagram on how this looks like (mostly) below userspace: Note how the kernel's HID devices are not exercised here at all because we parse the vendor report, create our own custom (separate) `uinput` device(s) and then basically re-implement the HID to evdev event mapping. This allows for great flexibility (and control, hence the vendor drivers are shipped this way) because any remapping can be done before you hit `uinput`. I don't immediately know whether OpenTabletDriver switches to firmware mode or maps the tablet mode but architecturally it doesn't make much difference. From a security perspective: having a userspace driver means you either need to run that driver daemon as root or (in the case of OpenTabletDriver at least) you need to allow `uaccess` to `/dev/uinput`, usually via `udev` rules. Once those are installed, anything can create `uinput` devices, which is a risk but how much is up for interpretation. [1] As is so often the case, even the intended state does not necessarily spark joy [2] Again, we're talking about the intended case here... [3] `fsvo` "no-one" [4] The `xf86-input-wacom` driver always initialises a separate eraser tool even if you never press that button [5] For historical reasons those are also multiplexed so getting `ABS_Z` on a device has different meanings depending on the tool currently in proximity [6] In our `udev-hid-bpf` BPFs we hide those devices so you really only get the correct event nodes, I'm not immediately sure what `hid-uclogic` does [7] At which point Pandora will once again open the box because most of the stack is not yet ready for non-Wacom tool ids

- [Natalie Vock: Fixing AMDGPU's VRAM management for low-end GPUs](#) (2026/04/09 00:00)

It may sound unbelievable to some, but not everyone has a datacenter beast with 128GB of VRAM shoved in their desktop PCs. Around the world people tell the tale of a particularly fierce group of Linux gamers: Those who dare attempt to play games with only 8 gigabytes of VRAM, or even less. Truly, it takes exceedingly strong resilience and determination to face the stutters and slowdowns bound to occur when the system starts running low on free VRAM. Carnage erupts inside the kernel driver as every application fights for as much GPU memory as it can hold on to. Any game caught up in this battle for resources will surely not leave unscathed. That is, until now. Because I fixed it. Q: I don't care about long-winded rants about Linux graphics drivers! Where do I get moar perf? A: You need some kernel patches as well as additional utilities to make use of the kernel capabilities properly. The simplest option is to use CachyOS (with KDE as your desktop). Their kernel includes the patches you need from version 7.0rc7-2 and up, and the userspace utilities are available in the package repositories. All you need to do is use CachyOS's 7.0rc7-2 kernel, install the packages called `dmemcg-booster` and `plasma-foreground-booster`, and you should be good to go. UPDATE: CachyOS's 6.19.12 kernel also includes this, now. No need to use the `-rc` kernel anymore. Q: I use another Arch-based distro! What now? The `dmemcg-booster` and `plasma-foreground-booster` utilities are available in the AUR as well (`plasma-foreground-booster` carries the package name `plasma-foreground-booster-dmemcg`), so you can install them from there. For the kernel side, you can either use the CachyOS kernel package on a non-CachyOS system by retrieving the package from their repository, or you can compile your own kernel. Installing `linux-dmemcg` from the AUR will compile the development branch I used to develop this. Being a development branch, this carries the risk of some stuff being broken, so install at your own risk! If you want to apply the kernel patches yourself, you need these six `.patch` files: Patch 1 Patch 2 Patch 3 Patch 4 Patch 5 Patch 6 I'm not sure how easily they apply on specific kernel versions, but feel free to leave a comment if you run into issues and I'll try to help out. Q: I don't use an Arch-based distro (or the instructions don't apply to me for some other reason)! What now? Maybe wait a bit. Eventually I'd expect this to trickle down into more distros. If I notice this work being packaged by other distros or being installable by other means, I will update this

blogpost. Q: I don't use KDE! What now? A: For games where you care about VRAM usage, you can use newer versions of gamescope. Newer versions of gamescope will also try to make use of these kernel capabilities, so running your games through that should be sufficient. You will still need the dmemcg-booster utility in any case. Q: I don't use systemd! What now? A: All the user-space utilities hard-depend on systemd. Without systemd, you'd need to write your own utilities that make use of my kernel patches. Something needs to manage cgroups in your system, and that something needs to enable the right cgroup controllers and set the right limits (see also the long-winded explanation about how this works). Q: I do care about long-winded rants about Linux graphics drivers! How does this work? Let's first look at what problems we actually run into when we have games running on GPUs with little VRAM. On a standard desktop system, the game won't be the only application that runs on the GPU at a time at all. If it's anything like my system, there's always at least one browser window with way too many tabs open, plus an assortment of apps (many of which are actually web apps running in their own browsers under the hood). All of this eats up quite a bit of VRAM, as well. To properly stress-test kernel memory management when working on this issue, I would go ahead and open up nearly every app with an integrated browser engine that I had installed. Viewed in `amdgpu_top`, the result of that looks something like this: Ouch, there goes 1/4 of VRAM. Now, let's try and launch Cyberpunk 2077 on top of that: As expected, the game uses a lot of VRAM (I cranked the settings really high). However, a lot of memory allocations also end up in a memory region referred to as "GTT". This is memory that is accessible by the GPU, but physically located in system RAM. From the GPU's point of view, system RAM memory has to be accessed over the PCI bus. Accessing memory over the PCI bus is typically really, really slow. On my system, instead of the 256GB/s bandwidth VRAM could provide, we're suddenly stuck with a meager 16GB/s at absolute maximum, paired with significantly worse latency. Some amount of memory landing in GTT is normal - many games will intentionally allocate memory in GTT because it is advantageous for some use cases. However, Cyberpunk 2077 allocates a fixed amount of around 650MB of memory in GTT. Instead, what happened here is that the game requested some memory allocations in VRAM, but somehow, they ended up in GTT instead! In kernel land, this process is referred to as eviction. The system in total tried to use more VRAM than there was available at all, so something had to give. Instead of telling the app that memory allocation failed (which would mean a near-certain application crash), the kernel decides to kick some memory out of VRAM to make everything fit. This degrades performance, but at least it allows every app to continue running. Nice! If only it would evict literally anything other than the game, which is the very thing that suffers the worst from having its memory evicted. Why in the world would it decide on that???? A brief history on kernel eviction policies Memory eviction and behavior under VRAM pressure are by no means new issues. Over the course of time, different approaches have tried tackling different associated issues, and those different approaches introduced new issues themselves. In the beginning, things worked rather simplistically: If applications wanted VRAM allocations, the user-mode driver would go to the kernel-mode driver and request VRAM memory. Save for some exceptional cases, that request would be granted, and that memory would be kept in VRAM. If another application requested VRAM allocations, and the memory was kicked out, the kernel driver would move the memory back into VRAM the next time work was submitted to the GPU using that memory. This worked quite horribly. Generally, two competing applications can be expected to roughly take turns executing GPU work - first one application submits work, then the other, then the first again, and so on. With that approach, memory would keep being moved back and forth after every single submission. One application gets kicked out and immediately moved back in, kicking the other out (which moves memory back in the next step). All this moving ended up with worse performance than if the memory had never been moved in the first place. The first bandaid solution was to rate-limit memory movement inside the kernel driver. Once the kernel driver moved enough memory within a specific time frame to trigger a limit, no more memory would be moved for some more time. This indeed reduced moves, but didn't do anything to fix the underlying issue of

repeated cyclic memory movements. Worse yet, repeatedly running into this ratelimit would introduce annoying jitters and stutters as the kernel driver rapidly alternated between moving memory and doing nothing. Eventually, to combat the still-existing overhead of repeatedly moving memory, user-mode drivers changed their allocation strategy. Instead of specifying VRAM as the only acceptable domain to place the allocation in, every VRAM allocation request would specify both "VRAM" and "GTT" as possible memory domains. The kernel would interpret this as VRAM being preferred, but if there was no space, GTT was an acceptable fallback and the kernel wouldn't try to kick out other VRAM memory to make space. This change entirely stopped the issue with memory repeatedly moving in and out of VRAM. However, if you squint your eyes a bit, you can see the kernel conceptually performing an eviction here, too. If there is no space in VRAM, the newly allocated memory is immediately evicted. This "eviction" is incredibly cheap to perform, since you don't actually need to move any memory, but the result is all the same: Memory that would ideally be in VRAM ends up in GTT. This case is what we run into in Cyberpunk 2077 above. At some point, VRAM is full, and new allocations done by the game go straight to GTT. Clearly, that is the wrong decision to make here. But being more aggressive wouldn't really work either - that was the approach before, and it was even worse. So what is the right decision here? Making the right decision is impossible. There is no single right decision to make here. Being aggressive is wrong, and not being aggressive is wrong, too. To be more specific, they're wrong in different cases. It makes complete sense for a game to be aggressive, but it makes no sense for random background apps to be equally aggressive. Random background apps should not be aggressive at all, but if the game backs off equally quickly, that doesn't help much either. The real problem is that to the kernel driver, all memory looks the same. The kernel doesn't know if it's dealing with a highly-important object from a game or a static image from a random web app running in the background - all it sees is a list of buffers. As long as all buffers look the same, it is impossible to have the same approach work well for every one of all the wildly different situations a driver may encounter. Enter cgroups cgroups are cool. They're super great at organizing random batches of processes into single organizational pieces. If you make a "compile job" cgroup and put the make process in it, all compiler processes it spawns will be part of that cgroup too. Don't want a big build hogging up all your RAM? Set a limit with the cgroup memory controller. Want to have some CPU time for other things? Just set a CPU limit with the cgroup cpu controller. It's great. You can have cgroup hierarchies too, and represent almost any kind of complex resource distribution you want. Luckily, systemd agrees that cgroups are cool. Every systemd unit is actually represented with its own cgroup, as well. And, as it happens, desktop environments will represent each desktop app as a systemd unit. How convenient! Complex resource distribution sounds exactly like the problems we're having in GPU driver land. If only someone wrote a cgroup controller operating on memory allocations from arbitrary devices such as GPUs... cgroups are a very clean solution for figuring out how relatively important GPU memory allocations are. Some time after Maarten Lankhorst from Intel initially wrote a cgroup controller managing GPU memory (initially only made for limiting how much VRAM one cgroup is allowed to consume), he pointed me to this work as a possible solution for the VRAM issues I was investigating. Eventually, this resulted in the dmem cgroup controller, written by Maarten, Maxime Ripard from Red Hat, and me. With the dmem cgroup controller, the kernel now learns about "memory protection". Memory being "protected" merely means that the kernel will go to significant lengths to avoid evicting that memory. For example, it may try to find memory from a different cgroup that is not protected and evict that instead. cgroups are all about resource partitioning, so for a cgroup, you can assign a "protection limit" - that is, if a cgroup's memory usage is below that limit, its memory is protected. As soon as it exceeds the limit, the memory ceases to be protected and can more easily be evicted. This roughly corresponds to the "more aggressive" and "less aggressive" behaviors we used to have, but now we can have some applications (=cgroups) that are more aggressive and some that are less aggressive. Precisely what we wanted! A note about my kernel patches The dmem cgroup controller has been upstream for a

while now, but for memory protection to work properly in gaming scenarios and such, you will likely still need my kernel patches. Remember how Cyberpunk 2077 ends up with its memory in GTT because the kernel driver sees that VRAM is exhausted and puts new memory in GTT right away? I argued this is conceptually equivalent to an eviction, but under the hood, this and real evictions that move existing memory from VRAM to GTT work very differently. Among other things, protection by dmem cgroups did not apply to these “evictions” - this is what my kernel patches fix. Without them, the kernel is still not aggressive enough even if there is protection, and allocations will still end up in GTT. User-space configuration Maybe the best thing about cgroups for VRAM management is that the prioritization is completely dynamic and configurable by userspace. Window managers can now determine whichever app is in the foreground and dedicate the highest priority to that app via its cgroup, completely without having to teach the GPU driver what a “window” or “foreground” is. For desktops, this is an important heuristic, but it’s totally not the kernel’s business to know the concept of a foreground app. I personally use KDE Plasma as my desktop environment, so I went looking for how such a thing could be integrated into Plasma. Lo and behold, it was already done! Plasma people already developed the ForegroundBooster utility that listens to which app is currently in the foreground, and tries to give it higher prioritization (in this case: wrt. CPU time) than other apps. This prioritization was also done via cgroups, so adding VRAM prioritization in my fork was pretty much a walk in the park. Except for one thing - the ForegroundBooster utility doesn’t manage cgroups and cgroup properties directly. systemd is responsible for managing cgroups, so ForegroundBooster just communicates with systemd to set the cgroup properties. That’s not too bad though, let’s just implement support for the dmem cgroup controller in systemd, right? Well, this is what I thought, too. But as I alluded to before, fixing VRAM management for gaming purposes is by far not the only possible purpose of dmem cgroups. There are quite a few other use cases that people are eyeing dmem cgroups for, and if I were to implement a systemd interface while only considering the gaming scenario, the other use cases run the risk of having to deal with a systemd interface that wasn’t designed with that use case in mind at all. So for now, a common systemd implementation seems mostly off-limits until the dust has settled some more. What do we do if we can’t tell systemd to do the thing we want? That’s right, we do it anyway, but behind systemd’s back. (Sorry, systemd.) This is what the final piece of the puzzle, dmemcg-booster does (safely and `▯blazingly fast▯`). After systemd constructs the cgroup hierarchy, dmemcg-booster goes over those cgroups and additionally enables the dmem controller on them, in order to activate the kernel functionality that ultimately allows for GPU memory protection on those cgroups. While at it, it also sets some settings in the cgroup hierarchy that allow the memory protection to kick in properly. Of course, this is a rather ugly stopgap. Once systemd gains proper support, you’d express all this with drop-in unit configurations, which is a much prettier approach. The dmemcg-booster utility is exclusively there to bridge the gap until that proper support happens. Conclusion With all the puzzle pieces finally in place, let’s repeat our test from before, launch a bunch of heavy apps, and then play Cyberpunk 2077 on top of that. How does it look now? GTT memory usage is now down to 650MB, i.e. only the memory that the game explicitly allocated in system RAM itself. Not a single piece of memory got spilled! Prioritization via cgroups now allows the game to use pretty much every last byte of VRAM for actual gaming purposes. It’s a bit hard to compare precise numbers on how the game performs, because the VRAM shortage slowly develops over time as you run around in the game, but the improvement should be obvious when comparing how games feel when you play them for a while. Instead of performance slowly degrading over time, games should perform much more stable - as long as the game itself doesn’t use more VRAM than you actually have. Generally, it seems like even modern games stay within a memory budget of ~8GB or a bit less, so if you have a GPU with 8GB of VRAM, you should be good to go with today’s games. More FAQ Which GPUs does this work with? Is it only AMD GPUs? Whether or not your GPU can benefit from it depends on the kernel driver - more specifically, whether it sets up the dmem cgroup controller. amdgpu and xe both have support for the dmem cgroup controller already. In theory,

Intel GPUs running the xe kernel driver should benefit as well, although I'm not sure anyone tested this yet. For nouveau, I have sent a patch for dmem cgroup support to the mailing lists. This patch is also included in my development branch, so if you use my AUR package it should work. In other cases, you will need to wait for the patch to be picked up by your distribution, or apply it yourself. The proprietary NVIDIA kernel modules do not support dmem cgroups yet, so this won't work there. Do iGPUs/APU systems benefit from this too? I don't actually know :) The main problem (system RAM being slower than dedicated VRAM) does not exist on integrated GPUs, because they use system RAM for everything - so effects will most likely be more limited than on dGPUs. Maybe it still has some benefit? It probably requires careful testing to find out. Are the kernel patches merged upstream? No, they haven't arrived in upstream Linux yet, but I've sent them to the mailing lists and things are in-progress. I'll update this when I know which mainline Linux release will pick these patches up.

- [Dave Airlie \(blogspot\): drm subsystem contributor numbers](#) (2026/04/01 20:59)

I'm doing a podcast recording this week, so I wanted to run some numbers so I could have some facts rather than feels. It turns out my feels were off by a factor of 3 or so. If asked, I've always said the contributor count to the drm subsystem is probably in the 100 or so developers per release cycle. Did the simplest: `git log --format='%aN' v6.14..v6.15 drivers/gpu/drm/ include/uapi/drm/ include/drm/ | sort -u | wc -l` iterated over a few kernel releases v6.15 326 v6.16 322 v6.17 300 v6.18 334 v6.19 332 v7.0-rc6 346 The number for the complete kernel in those scenarios are ~2000 usually, which means drm subsystem has around 15-16% of the kernel contributors. I'm a bit spun out, that's quite a lot of people. I think I'll blame Sima for it. This also explains why I'm a bit out of touch with the process problems other maintainers have, and when I say stuff like a lot of workflows don't scale, this is what I mean.

- [Sebastian Wick: Three Little Rust Crates](#) (2026/03/27 00:15)

I published three Rust crates: `name-to-handle-at`: Safe, low-level Rust bindings for Linux `name_to_handle_at` and `open_by_handle_at` system calls `pidfd-util`: Safe Rust wrapper for Linux process file descriptors (`pidfd`) `listen-fds`: A Rust library for handling `systemd` socket activation They might seem like rather arbitrary, unconnected things - but there is a connection! `systemd` socket activation passes file descriptors and a bit of metadata as environment variables to the activated process. If the activated process exec's another program, the file descriptors get passed along because they are not `CLOEXEC`. If that process then picks them up, things could go very wrong. So, the activated process is supposed to mark the file descriptors `CLOEXEC`, and unset the socket activation environment variables. If a process doesn't do this for whatever reason however, the same problems can arise. So there is another mechanism to help prevent it: another bit of metadata contains the PID of the target. Processes can check it against their own PID to figure out if they were the target of the activation, without having to depend on all other processes doing the right thing. PIDs however are racy because they wrap around pretty fast, and that's why nowadays we have `pidfds`. They are file descriptors which act as a stable handle to a process and avoid the ID wrap-around issue. Socket activation with `systemd` nowadays also passes a `pidfd` ID. A `pidfd` ID however is not the same as a `pidfd` file descriptor! It is the 64 bit inode of the `pidfd` file descriptor on the `pidfd` filesystem. This has the advantage that `systemd` doesn't have to install another file descriptor in the target process which might not get closed. It can just put the `pidfd` ID number into the `$LISTEN_PIDFDID` environment variable. Getting the inode of a file descriptor doesn't sound hard. `fstat(2)` fills out `struct stat` which has the `st_ino` field. The problem is that it has a type of `ino_t`, which is 32 bits on some systems so we might end up with a process identifier which wraps around pretty fast again. We can however use the `name_to_handle` syscall on the `pidfd` to get a `struct file_handle` with a `f_handle` field. The man page helpfully says that "the caller should treat the `file_handle` structure as an opaque data type". We're going to ignore that, though, because at least on the `pidfd` filesystem, the first 64 bits are the 64 bit inode. With `systemd` already

depending on this and the kernel rule of “don’t break user-space”, this is now API, no matter what the man page tells you. So there you have it. It’s all connected. Obviously both `pidfds` and `name_to_handle` have more exciting uses, many of which serve my broader goal: making Varlink services a first-class citizen. More about that another time.

- [Lennart Poettering: Mastodon Stories for systemd v260](#) (2026/03/26 23:00)

On March 17 we released systemd v260 into the wild. In the weeks leading up to that release (and since then) I have posted a series of serieses of posts to Mastodon about key new features in this release, under the `#systemd260` hash tag. In case you aren't using Mastodon, but would like to read up, here's a list of all 21 posts: Post #1: NvPCR Measurements for Activated DDIs Post #2: Varlink Transport Plugins Post #3: Well-Known Varlink Services Post #4: `.mstack` Overlay Mount Stacks Post #5: `RefreshOnReload=` in Service Units Post #6: `FANCY_NAME=` in `/etc/os-release` Post #7: `BindNetworkInterface=` in Service Units Post #8: `importctl` pull-oci for Acquiring OCI Containers Post #9: `systemd-report` and Metrics API Post #10: `udev's` `tpm2_id` built-in and the TPM2 Quirks Database Post #11: `Devicetree/CHID` Database Post #12: Varlink IPC for `systemd-networkd` Post #13: `systemd-vmspawn` knows `--ephemeral` now Post #14: `systemd-loginds's` `xaccess` Concept Post #15: Unprivileged Portable Services Post #16: Image Policy Improvements Post #17: LUKS Volume Key Fixation Post #18: Journal Varlink Access Post #19: Nested UID Range Delegation Post #20: `PrivateUsers=managed` Post #21: `bootctl` install as Varlink API I intend to do a similar series of serieses of posts for the next systemd release (v261), hence if you haven't left tech Twitter for Mastodon yet, now is the opportunity. My series for v261 will begin in a few weeks most likely, under the `#systemd261` hash tag. In case you are interested, here is the corresponding blog story for systemd v259, here for v258, here for v257, and here for v256.

- [Erik Faye-Lund: Open Source and AI](#) (2026/03/26 16:30)

More and more frequently, I get asked about my stance on AI in the context of programming. This is my attempt to summarize my stance for those who wonder. This is a blog post that I don’t want to write, but some recent developments have more or less forced my hand here. I would have preferred to keep pretending that I’m neutral in the issue, and just hoping that the problem goes away. But that doesn’t seem to be happening. I’m probably not the most qualified person to write about this, I’m sure you can find better informed articles out there. These are my personal opinions, and not those of my employer, their customers or any other of my affiliates. Take them with a pinch of salt, and feel free to disagree. On a general note, I’m very reluctant to tell people how they should behave. But in this case, I’ve decided to do just that. I hope it’s clear why from the context. Similarly, I would caution every reader to be skeptical of anyone who claims to know what the future holds, me included. People often predict the future that benefits themselves the most. There’s a few times I make some predictions in this post. Those are just predictions, and I might very well be wrong. A final warning; this is a long post, so set aside some time. I’ve tried to limit the scope somewhat to mostly cover topics concerning open source development, but I sometimes end up discussing wider issues. This is simply because I don’t feel like I can ignore these. But yeah, let’s start close to home here... Licensing issues - The “plagiarism machine” Currently, the legal status of AI generated code is still far from clear. Is it derivative work of all of the training data or not? It sometimes can be, but it depends on a lot of factors. This is a critical issue to open source; I can’t submit code somewhere that I don’t know the origin of and potentially is license-incompatible with the upstream project. This isn’t just theoretical, Copilot has been known to output Quake 3 source code with the wrong license. It doesn’t matter if the people with the most to lose from AI output counting as derivative works keeps insisting not to worry. The US supreme court recently made it clear that AI generated code isn’t even copyright-able. A human needs to write the code for copyright to be granted. But with using AI tools, we’re blurring the line, making it hard or even impossible to tell what is written by a human and what isn’t. I also doubt that a single or a handful

of lawsuits is going to be enough to settle this. We're working in a global ecosystem, and there's potentially hundreds of jurisdictions that might have to rule, and just as many subtleties to take into account before we have a good understanding around this. It's going to take a long time to find out. But even if this wasn't an issue, does that really mean we should use AI? Open source software is inherently political, especially when it comes to licensing. I tend to find something not being illegal to be a terribly low standard to have. It should IMO also be the right thing to do. This brings us to the other issues... The cost of AI scraping As open source developers, it's important for our infrastructure to be publicly available to everyone. In recent times, AI scrapers have started taking advantage of this, and are now aggressively scraping all content on open source support infrastructure so they can train their models. These scrapers often ignore robots.txt directives, and sometimes even use randomized, residential IP addresses, making it hard or impossible to effectively block them. All of this has a major financial impact on open source projects. It's not unusual to see over 90% of traffic provably coming from AI scrapers. As a result, the open source community has had to introduce barriers, like Anubis, which slows down initial page-loads. Since Anubis is based on proof-of-work, it means people with slow computers can no longer reach our infrastructure. And I can't reliably browse our GitLab instance from my phone on the bus to work. While the latter is a minor annoyance, the former is a real problem for inclusivity. And because our infrastructure is so heavily affected by this, it feels deeply problematic to me if we use (and pay for) the tools that are built on this behavior. That would be rewarding the behavior. We should vote with our wallets here, and in this case this means to not pay them. Maintainability issues Another issue is that code needs to be understood and maintained in the long term. For this to work well, we need to be able to reach out to the people that wrote the code and get input on what led to a decision. Obviously, that's not always possible, but with AI this is almost never possible. The context is lost, and so is all the insight. Asking an AI again about the same code might lead to completely different reasoning, and miss crucial details. The project I'm mostly working on, Mesa 3D, is also arguably critical infrastructure for a lot of computer systems around the world. We need to be lean towards being conservative rather than experimental when building these kinds of systems. The junior problem Another related issue is that AI technology tends to be used to take over more "junior" tasks, but the result of this is likely to be that we end up hiring and mentoring fewer junior developers. This will lead us to having fewer competent senior developers in the future. Interacting with an AI isn't going to gradually make the AI learn and become more senior, unlike with a junior developer. AIs learn from training, not queries. Mentoring junior developers builds trust, which makes the interactions worthwhile also for the senior. In my experience, interactions with AIs are little other than frustration that never improves. And because working with the AI doesn't build any meaningful trust, the AIs will always need guard-rails to prevent disasters. A future where we develop software with few to no human developers (junior or senior) sounds scary to me, but that's where this path leads. Environmental issues Building and running these huge data-centers is extremely resource heavy. Some of these resources are resources we all have to share on this planet, like water, electricity and rare earth minerals. This is taking a toll on our planet and everything living on it. I feel like this point got a lot less attention recently than it used to, but it hasn't really been solved. Instead, the AI giants have just doubled down on wanting to consume all the resources they feel they need, without regard for the planet or people living on it. They are far from truthful about how bad this is, and try to prevent us from knowing just how bad it is. The truth is that AI-type solutions are almost always one of the most resource intensive solutions to problems possible. And right now we're being told that we should use it for all problems. This is a recipe for disaster, nothing less. And it seems like there's nothing being done on this front. The big AI companies are just slowly boiling the ocean, hoping that we don't notice or that we forget. I haven't forgotten. Economical issues A secondary effect of building all these data-centers is that demand for a bunch of resources goes up, and so does the price. This affects everyone. We're not just seeing electricity and water being more scarce, we're also seeing memory and storage prices spiking hard as well.

Forget about buying a new GPU, and just generally wait a couple of years with buying a new computer, or really any new gadgets. How can this possibly not lead to a recession if things are allowed to continue? And then we have the blatant circular economy that the big AI players are doing to try to convince the market this is actually profitable. In reality, very little actual money is changing hands, they're mostly just making promises to buy tech from each other in the future... Which brings us to the big one... The bubble Yeah, so it seems very likely we're currently in a bubble. We've been for a while, and this bubble is going to pop. The question is when and how. Don't get me wrong; not all bubbles pop and erase everything with it. The dot-com bubble took years to pop, and we still have computers and the internet and all that jazz. But we're currently overspending on infrastructure, and the companies selling that are currently raking in, and they are trying hard to make us all dependent on their technology. For the last few years, the AI industry has slurped up most of the traditional technology investment capital available. The investors seem less and less interested in investing more money into the AI industry, and want return on their investments instead. So they have started turning to things like pension funds. If they get away with this, everyone is going to pay for this, regardless of their involvement in AI. We've already been seeing the idea of "too big to fail" being thrown out there, mirroring what happened in the subprime mortgage crisis. We should, as a society refrain from letting them do this. These problems are caused by the AI industry, not by us consumers. We shouldn't be the ones to bail them out when the time comes. OpenAI's CFO has already suggested that the U.S. government should provide a \$1.4 trillion "safety net" for AI investments, and while Sam Altman since has walked that back after public outcry, this shows that these companies are already thinking along those lines. Keeping the brain active On a more personal note, it's kinda undeniable; I'm getting older, and part of getting old means that I need to spend more time actively thinking about things to keep up. Letting an AI take over the wheels, even just for the boring bits doesn't help me, it only makes this worse. Keeping the brain sharp requires work, not assistance. In fact, I often feel like I learn something useful, even when I do mundane tasks. Asking an LLM to write up a python script for me to do something robs me of learning in the process of doing it myself. Add that to the data that suggests that we actually get less productive by using AI (while thinking we're more productive), makes this all very unappealing to me. My brain is my most important tool, and I'm not going to risk it because tech CEOs are yelling at the world that they need to use AI to prevent a recession. Conclusion You might have noticed that I don't really address the technical abilities of current AI technologies in this post. The reason is that I don't feel like I need to; it's kinda irrelevant. I think the moral arguments against using AI for open source development are just too large to ignore. In fact, just the licensing and environmental issues alone would probably have been enough for me to draw a hard line in the sand here: Using AI for open source projects is in my opinion immoral, and I will not be using it. I do not condone others using AI for anything in the open source ecosystem either. Using it is simply detrimental to our values and directly harms our community. If you're currently playing around with AI out of curiosity for open source projects, I would like to ask you to reconsider. If you're working in a company that's encouraging AI usage, I would like to ask you to speak up against it. If you are involved in policy decisions for open source projects, I would like to encourage you to try your best to discourage AI adoption within those projects. Our entire ecosystem is on the line here. Not just the open source ecosystem, but the entire, global ecosystem. And I feel there's not enough voices speaking up about it. Make your voice heard! Allow yourself to be angry; there's enough nonsense going out there! We need to stop this madness.

- [Christian Schaller: Using AI to create some hardware tools and bring back the past](#) (2026/03/23 16:07)

As I talked about in a couple of blog posts now I been working a lot with AI recently as part of my day to day job at Red Hat, but also spending a lot of evenings and weekend time on this (sorry kids pappa has switched to 1950's mode for now). One of the things I spent time on is trying to figure out what the limitations of AI models are and what kind of use they can have for Open Source developers. One thing to mention before I

start talking about some of my concrete efforts is that I more and more come to conclude that AI is an incredible tool to hypercharge someone in their work, but I feel it tend to fall short for fully autonomous systems. In my experiments AI can do things many many times faster than you ordinarily could, talking specifically in the context of coding here which is what is most relevant for those of us in the open source community. So one annoyance I had for years as a Linux user is that I get new hardware which has features that are not easily available to me as a Linux user. So I have tried using AI to create such applications for some of my hardware which includes an Elgato Light and a Dell Ultrasharp Webcam. I found with AI and this is based on using Google Gemini, Claude Sonnet and Opus and OpenAI codex, they all required me to direct and steer the AI continuously, if I let the AI just work on its own, more often than not it would end up going in circles or diverging from the route it was supposed to go, or taking shortcuts that makes wanted output useless. On the other hand if I kept on top of the AI and intervened and pointed it in the right direction it could put together things for me in very short time spans. My projects are also mostly what I would describe as end leaf nodes, the kind of projects that already are 1 person projects in the community for the most part. There are extra considerations when contributing to bigger efforts, and I think a point I seen made by others in the community too is that you need to own the patches you submit, meaning that even if an AI helped your write the patch you still need to ensure that what you submit is in a state where it can be helpful and is merge-able. I know that some people feel that means you need be capable of reviewing the proposed patch and ensuring its clean and nice before submitting it, and I agree that if you expect your patch to get merged that has to be the case. On the other hand I don't think AI patches are useless even if you are not able to validate them beyond 'does it fix my issue'. My friend and PipeWire maintainer Wim Taymans and I was talking a few years ago about what I described at the time as the problem of 'bad quality patches', and this was long before AI generated code was a thing. Wim response to me which I often thought about afterwards was "a bad patch is often a great bug report". And that would hold true for AI generated patches to. If someone makes a patch using AI, a patch they don't have the ability to code review themselves, but they test it and it fixes their problem, it might be a good bug report and function as a clearer bug report than just a written description by the user submitting the report. Of course they should be clear in their bug report that they don't have the skills to review the patch themselves, but that they hope it can be useful as a tool for pinpointing what isn't working in the current codebase. Anyway, let me talk about the projects I made. They are all found on my personal website linuxrising.org a website that I also used AI to update after not having touched the site in years.

Elgato Light GNOME Shell extension Elgato Light GNOME Shell extension The first project I worked on is a GNOME Shell extension for controlling my Elgato Key Wifi Lamp. The Elgato lamp is basically meant for podcasters and people doing a lot of video calls to be able to easily configure light in their room to make a good recording. The lamp announces itself over mDNS, and thus can be controlled via Avahi. For Windows and Mac the vendor provides software to control their lamp, but unfortunately not for Linux. There had been GNOME Shell extensions for controlling the lamp in the past, but they had not been kept up to date and their feature set was quite limited. Anyway, I grabbed one of these old extensions and told Claude to update it for latest version of GNOME. It took a few iterations of testing, but we eventually got there and I had a simple GNOME Shell extension that could turn the lamp off and on and adjust hue and brightness. This was a quite straightforward process because I had code that had been working at some point, it just needed some adjustments to work with current generation of GNOME Shell. Once I had the basic version done I decided to take it a bit further and try to recreate the configuration dialog that the windows application offers for the full feature set which took me quite a bit of back and forth with Claude. I found that if I ask Claude to re-implement from a screenshot it recreates the functionality of the user interface first, meaning that it makes sure that if the screenshot has 10 buttons, then you get a GUI with 10 buttons. You then have to iterate both on the UI design, for example telling Claude that I want a dark UI style to match the GNOME Shell, and then I also had

to iterate on each bit of functionality in the UI. Like most of the buttons in the UI didn't really do anything from the start, but when you go back and ask Claude to add specific functionality per button it is usually able to do so. Elgato Light Settings Application So this was probably a fairly easy thing for the AI because all the functionality of the lamp could be queried over Avahi, there was no 'secret' USB registers to be set or things like that. Since the application was meant to be part of the GNOME Shell extension I didn't want to have any dependency requirements that the Shell extension itself didn't have, so I asked Claude to make this application in JavaScript and I have to say so far I haven't seen any major differences in terms of the AIs ability to generate different languages. The application now reproduce most of the functionality of the Windows application. Looking back I think it probably took me a couple of days in total putting this tool together. Dell Ultrasharp Webcam 4K Dell UltraSharp 4K settings application for Linux The second application on the list is a controller application for my Dell UltraSharp Webcam 4K UHD (WB7022). This is a high end Webcam I that have been using for a while and it is comparable to something like the Logitech BRIO 4K webcam. It has mostly worked since I got it with the generic UVC driver and I been using it for my Google Meetings and similar, but since there was no native Linux control application I could not easily access a lot of the cameras features. To address this I downloaded the windows application installer and installed it under Windows and then took a bunch of screenshots showcasing all features of the application. I then fed the screenshots into Claude and told it I wanted a GTK+ version for Linux of this application. I originally wanted to have Claude write it in Rust, but after hitting some issues in the PipeWire Rust bindings I decided to just use C instead. I took me probably 3-4 days with intermittent work to get this application working and Claude turned out to be really good and digging into Windows binaries and finding things like USB property values. Claude was also able to analyze the screenshots and figure out the features the application needed to have. It was a lot of trial and error writing the application, but one way I was able to automate it was by building a screenshot option into the application, allowing it to programmatically take screenshots of itself. That allowed me to tell Claude to try fixing something and then check the screenshot to see if it worked without me having to interact with the prompt. Also to get the user interface looking nicer, once I had all the functionality in I asked Claude to tweak the user interface to follow the guidelines of the GNOME Human Interface Guidelines, which greatly improved the quality of the UI. At this point my application should have almost all the features of the Windows application. Since it is using PipeWire underneath it is also tightly integrated with the PipeWire media graph, allowing you to see it connect and work with your application in PipeWire patchbay applications like Helvum. The remaining features are software features of Dell's application, like background removal and so on, but I think that if I decided to implement that it should be as a standalone PipeWire tool that can be used with any camera, and not tied to this specific one. Red Hat Planet The application shows the worlds Red Hat offices and include links to latest Red Hat news. The next application on my list is called Red Hat Planet. It is mostly a fun toy, but I made it to partly revisit the Xtraceroute modernisation I blogged about earlier. So as I mentioned in that blog, Xtraceroute while cute isn't really very useful IMHO, since the way the modern internet works rarely have your packets jump around the world. Anyway, as people pointed out after I posted about the port is that it wasn't an actual Vulkan application, it was a GTK+ application using the GTK+ Vulkan backend. The Globe animation itself was all software rendered. I decided if I was going to revisit the Vulkan problem I wanted to use a different application idea than traceroute. The idea I had was once again a 3D rendered globe, but this one reading the coordinates of Red Hats global offices from a file and rendering them on the globe. And alongside that provide clickable links to recent Red Hat news items. So once again maybe not the worlds most useful application, but I thought it was a cute idea and hopefully it would allow me to create it using actual Vulkan rendering this time. Creating this turned out to be quite the challenge (although it seems to have gotten easier since I started this effort), with Claude Opus 4.6 being more capable at writing Vulkan code than Claude Sonnet, Google Gemini or OpenAI Codex was when I started trying to create this application. When I

started this project I had to keep extremely close tabs on the AI and what it was doing in order to force it to keep working on this as a Vulkan application, as it kept wanting to simplify with Software rendering or OpenGL and sometimes would start down that route without even asking me. That hasn't happened more recently, so maybe that was a problem of AI of 5 Months ago. I also discovered as part of this that rendering Vulkan inside a GTK4 application is far from trivial and would ideally need the GTK4 developers to create such a widget to get rendering timings and similar correct. It is one of the few times I have had Claude outright say that writing a widget like that was beyond its capabilities (haven't tried again so I don't know if I would get the same response today). So I started moving the application to SDL3 first, which worked as I got a spinning globe with red dots on, but came with its own issues, in the sense that SDL is not a UI toolkit as such. So while I got the globe rendered and working the AU struggled badly with the news area when using SDL. So I ended up trying to port the application to Qt, which again turned out to be non-trivial in terms of how much time it took with trial and error to get it right. I think in my mind I had a working globe using Vulkan, how hard could it be to move it from SDL3 to Qt, but there was a million rendering issues. In fact I ended up using the Qt Vulkan rendering example as a starting point in the end and then 'porting' the globe over bit by bit, testing it for each step, to finally get a working version. The current version is a Vulkan+Qt app and it basically works, although it seems the planet is not spinning correctly on AMD systems at the moment, while it seems to work well on Intel and NVIDIA systems. WMDock WmDock fullscreen with config application. This project came out of a chat with Matthias Clasen over lunch where I mused about if Claude would be able to bring the old Window Maker dockapps to GNOME and Wayland. Turns out the answer is yes although the method of doing so changed as I worked on it. My initial thought was for Claude to create a shim that the old dockapps could be compiled against, without any changes. That worked, but then I had a ton of dockapps showing up in things like the alt+tab menu. It also required me to restart my GNOME Shell session all the time as I was testing the extension to house the dockapps. In the end I decided that since a lot of the old dockapps don't work with modern Linux versions anyway, and thus they would need to be actively ported, I should accept that I ship the dockapps with the tool and port them to work with modern linux technologies. This worked well and is what I currently have in the repo, I think the wildest port was porting the old dockapp webcam app from V4L1 to PipeWire. Although updating the soundcontroller from ESD to PulesAudio was also a generational jump. XMMS resuscitated XMMS brought back to life So the last effort I did was reviving the old XMMS media player. I had tried asking Claude to do this for Months and it kept failing, but with Opus 4.6 it plowed through it and had something working in a couple of hours, with no input from me beyond kicking it off. This was a big lift, moving it from GTK2 and Esound, to GTK4, GStreamer and PipeWire. One thing I realized is that a challenge with bringing an old app back is that since keeping the themeable UI is a big part of this specific application adding new features is a little kludgy. Anyway I did set it up to be able to use network speakers through PipeWire and also you can import your Spotify playlists and play those, although you need to run the Spotify application in the background to be able to play sound on your local device. Monkey Bubble Monkey Bubble was a game created in the heyday of GNOME 2 and while I always thought it was a well made little game it had never been updated to newer technologies. So I asked Claude to port it to GTK4 and use GStreamer for audio. This port was fairly straightforward with Claude having little problems with it. I also asked Claude to add highscores using the libmanette library and network game discovery with Avahi. So some nice little improvements. All the applications are available either as Flatpaks or Fedora RPMS, through the gitlab project page, so I hope people enjoy these applications and tools. And enjoy the blasts from the past as much as I did. Worries about Artificial Intelligence When I speak to people both inside Red Hat and outside in the community I often come across negativity or even sometimes anger towards Artificial Intelligence in the coding space. And to be clear I do worry about where things could be heading and how it will affect my livelihood too, so I am not unsympathetic to those worries at all. I probably worry about these things at least a

few times a day. At the same time I don't think we can hide from or avoid this change, it is happening with or without us. We have to adapt to a world where this tool exists, just like our ancestors have adapted to jobs changing due to industrialization and science before. So do I worry about the future, yes I do. Do I worry about how I might personally get affected by this? yes, I do. Do I worry about how society might change for the worse due to this? yes, I do. But I also remind myself that I don't know the future and that people have found ways to move forward before and society has survived and thrived. So what I can control is that I try to be on top of these changes myself and take advantage of them where I can and that is my recommendation to the wider open source community on this too. By leveraging them to move open source forward and at the same time trying to put our weight on the scale towards the best practices and policies around Artificial Intelligence. The Next Test and where AI might have hit a limit for me. So all these previous efforts did teach me a lot of tricks and helped me understand how I can work with an AI agent like Claude, but especially after the success with the webcam I decided to up the stakes and see if I could use Claude to help me create a driver for my Plustek OpticFilm 8200i scanner. So I have zero background in any kind of driver development and probably less than zero in the field of scanner driver specifically. So I ended up going down a long row of deadends on this journey and I to this day has not been able to get a single scan out of the scanner with anything that even remotely resembles the images I am trying to scan. My idea was to have Claude analyse the Windows and Mac driver and build me a SANE driver based on that, which turned out to be horribly naive and lead nowhere. One thing I realized is that I would need to capture USB traffic to help Claude contextualize some of the findings it had from looking at the Windows and Mac drivers. I started out with Wireshark and feeding Claude with the Wireshark capture logs. Claude quite soon concluded that the Wireshark logs wasn't good enough and that I needed lower level traffic capture. Buying a USB packet analyzer isn't cheap so I had the idea that I could use one of the ARM development boards floating around the house as a USB relay, allowing me to perfectly capture the USB traffic. With some work I did manage to set up my LibreComputer Solitude AML-S905D3-CC arm board going and setting it in device mode. I also had a usb-relay daemon going on the board. After a lot of back and forth, and even at one point trying to ask Claude to implement a missing feature in the USB kernel stack, I realized this would never work and I ended up ordering a Beagle USB 480 USB hardware analyzer. At about the same time I came across the chipset documentation for the Genesys Logic GL845 chip in the scanner. I assumed that between my new USB analyzer and the chipset docs this would be easy going from here on, but so far no. I even had Claude decompile the windows driver using ghidra and then try to extract the needed information needed from the decompiled code. I bought a network controlled electric outlet so that Claude can cycle the power of the scanner on its own. So the problem here is that with zero scanner driver knowledge I don't even know what I should be looking for, or where I should point Claude to, so I kept trying to brute force it by trial and error. I managed to make SANE detect the scanner and I managed to get motor and lamp control going, but that is about it. I can hear the scanner motor running and I ask for a scan, but I don't know if it moves correctly. I can see light turning on and off inside the scanner, but I once again don't know if it is happening at the correct times and correct durations. And Claude has of course no way of knowing either, relying on me to tell it if something seems like it has improved compared to how it was. I have now used Claude to create two tools for Claude to use, once using a camera to detect what is happening with the light inside the scanner and the other recording sound trying to compare the sound this driver makes compared to the sounds coming out when doing a working scan with the MacOS X application. I don't know if this will take me to the promised land eventually, but so far I consider my scanner driver attempt a giant failure. At the same time I do believe that if someone actually skilled in scanner driver development was doing this they could have guided Claude to do the right things and probably would have had a working driver by now. So I don't know if I hit the kind of thing that will always be hard for an AI to do, as it has to interact with things existing in the real world, or if newer versions of Claude, Gemini or Codex will suddenly get past a threshold

and make this seem easy, but this is where things are at for me at the moment.

- [Alberto Ruiz: Booting with Rust: Chapter 3](#) (2026/03/18 04:52)

In Chapter 1 I gave the context for this project and in Chapter 2 I showed the bare minimum: an ELF that Open Firmware loads, a firmware service call, and an infinite loop. That was July 2024. Since then, the project has gone from that infinite loop to a bootloader that actually boots Linux kernels. This post covers the journey. The filesystem problem The Boot Loader Specification expects BLS snippets in a FAT filesystem under `loaders/entries/`. So the bootloader needs to parse partition tables, mount FAT, traverse directories, and read files. All `#[no_std]`, all big-endian PowerPC. I tried writing my own minimal FAT32 implementation, then integrating `simple-fatfs` and `fatfs`. None worked well in a freestanding big-endian environment. Hadris The breakthrough was `hadris`, a `no_std` Rust crate supporting FAT12/16/32 and ISO9660. It needed some work to get going on PowerPC though. I submitted fixes upstream for: `thiserror` pulling in `std`: default features were not disabled, preventing `no_std` builds. Endianness bug: the FAT table code read cluster entries as native-endian `u32`. On x86 that's invisible; on big-endian PowerPC it produced garbage cluster chains. Performance: every cluster lookup hit the firmware's block I/O separately. I implemented a 4MiB readahead cache for the FAT table, made the window size parametric at build time, and improved `read_to_vec()` to coalesce contiguous fragments into a single I/O. This made kernel loading practical. All patches were merged upstream. Disk I/O Hadris expects Read + Seek traits. I wrote a PROMDisk adapter that forwards to OF's read and seek client calls, and a Partition wrapper that restricts I/O to a byte range. The filesystem code has no idea it's talking to Open Firmware. Partition tables: GPT, MBR, and CHRP PowerVM with modern disks uses GPT (via the `gpt-parser` crate): a PReP partition for the bootloader and an ESP for kernels and BLS entries. Installation media uses MBR. I wrote a small `mbr-parser` subcrate using explicit-endian types so little-endian LBA fields decode correctly on big-endian hosts. It recognizes FAT32, FAT16, EFI ESP, and CHRP (type 0x96) partitions. The CHRP type is what CD/DVD boot uses on PowerPC. For ISO9660 I integrated `hadris-iso` with the same Read + Seek pattern. Boot strategy? Try GPT first, fall back to MBR, then try raw ISO9660 on the whole device (CD-ROM). This covers disk, USB, and optical media. The firmware allocator wall This cost me a lot of time. Open Firmware provides `claim` and `release` for memory allocation. My initial approach was to implement Rust's `GlobalAlloc` by calling `claim` for every allocation. This worked fine until I started doing real work: parsing partitions, mounting filesystems, building vectors, sorting strings. The allocation count went through the roof and the firmware started crashing. It turns out SLOF has a limited number of tracked allocations. Once you exhaust that internal table, `claim` either fails or silently corrupts state. There is no documented limit; you discover it when things break. The fix was to `claim` a single large region at startup (1/4 of physical RAM, clamped to 16-512 MB) and implement a free-list allocator on top of it with block splitting and coalescing. Getting this right was painful: the allocator handles arbitrary alignment, coalesces adjacent free blocks, and does all this without itself allocating. Early versions had coalescing bugs that caused crashes which were extremely hard to debug - no debugger, no backtrace, just writing strings to the OF console on a 32-bit big-endian target. And the kernel boots! March 7, 2026. The commit message says it all: "And the kernel boots!" The sequence: BLS discovery: walk `loaders/entries/*.conf`, parse into `BLSEntry` structs, filter by architecture (`ppc64le`), sort by version using `rpmvercmp`. ELF loading: parse the kernel ELF, iterate `PT_LOAD` segments, `claim` a contiguous region, copy segments to their virtual address offsets, zero BSS. `initrd`: `claim` memory, load the `initramfs`. `bootargs`: set `/chosen/bootargs` via `setprop`. Jump: inline assembly trampoline - `r3=initrd` address, `r4=initrd` size, `r5=OF` client interface, branch to kernel: `core::arch::asm!("mr 7, 3", // save of_client "mr 0, 4", // r0 = kernel_entry "mr 3, 5", // r3 = initrd_addr "mr 4, 6", // r4 = initrd_size "mr 5, 7", // r5 = of_client "mtctr 0", "bctr", in("r3") of_client, in("r4") kernel_entry, in("r5") initrd_addr as usize, in("r6") initrd_size as usize, options(nostack, noreturn))` One gotcha: do NOT close `stdout/stdin` before jumping. On some firmware, closing them corrupts `/chosen` and the kernel hits a machine check. We also skip

calling exit or release - the kernel gets its memory map from the device tree and avoids claimed regions naturally. The boot menu I implemented a GRUB-style interactive menu: Countdown: boots the default after 5 seconds unless interrupted. Arrow/PgUp/PgDn/Home/End navigation. ESC: type an entry number directly. e: edit the kernel command line with cursor navigation and word jumping (Ctrl+arrows). This runs on the OF console with ANSI escape sequences. Terminal size comes from OF's Forth interpret service (`#columns / #lines`), with serial forced to 80x24 because SLOF reports nonsensical values. Secure boot (initial, untested) IBM POWER has its own secure boot: the `ibm,secure-boot` device tree property (0=disabled, 1=audit, 2=enforce, 3=enforce+OS). The Linux kernel uses an appended signature format - PKCS#7 signed data appended to the kernel file, same format GRUB2 uses on IEEE 1275. I wrote an appended-sig crate that parses the appended signature layout, extracts an RSA key from a DER X.509 certificate (compiled in via `include_bytes!`), and verifies the signature (SHA-256/SHA-512) using the RustCrypto crates, all `no_std`. The unit tests pass, including an end-to-end sign-and-verify test. But I have not tested this on real firmware yet. It needs a PowerVM LPAR with secure boot enforced and properly signed kernels, which QEMU/SLOF cannot emulate. High on my list. The `ieee1275-rs` crate The crate has grown well beyond Chapter 2. It now provides: `claim/release`, the custom heap allocator, device tree access (`finddevice`, `getprop`, `instance-to-package`), block I/O, console I/O with `read_stdin`, a Forth interpret interface, milliseconds for timing, and a `GlobalAlloc` implementation so `Vec` and `String` just work. Published on crates.io at github.com/rust-osdev/ieee1275-rs. What's next I would like to test the Secure Boot feature on an end to end setup but I have not gotten around to request access to a PowerVM PAR. Beyond that I want to refine the menu. Another idea would be to perhaps support the equivalent of the Unified Kernel Image using ELF. Who knows, if anybody finds this interesting let me know! The source is at the `powerpc-bootloader` repository. Contributions welcome, especially from anyone with POWER hardware access.

- [Sebastian Wick: Redefining Content Updates in Wayland](#) (2026/03/10 22:56)

The Wayland core protocol has described surface state updates the same way since the beginning: requests modify pending state, commits either apply that state immediately or cache it into the parent for synchronized subsurfaces. Compositors implemented this model faithfully. Then things changed. Buffer Readiness and Compositor Deviation The problem emerged from GPU work timing. When a client commits a surface with a buffer, that buffer might still have GPU rendering in progress. If the compositor applies the commit immediately, it would display incomplete content—glitches. If the compositor submits its own GPU work with a dependency on the unfinished client work, it risks missing the deadlines for the next display refresh cycles and even worse stalling in some edge cases. To get predictable timing, the compositor needs to defer applying commits until the GPU work finishes. This requires tracking readiness constraints on committed state. Mutter was the first compositor to address this by implementing constraints and dependency tracking of content updates internally. Instead of immediately applying or caching commits, Mutter queued the changes in what we now call content updates, and only applied them when ready. Critically, this was an internal implementation detail. From the client's perspective, the protocol semantics remained unchanged. Mutter had deviated from the implementation model implied by the specification while maintaining the observable behavior. New Protocols on Unstable Foundations When we wanted better frame timing control and a proper FIFO presentation modes on Wayland, we suddenly required explicit queuing of content updates to describe the behavior of the protocols. You can't implement FIFO and scheduling of content updates without a queue, so both the `fifo` and `commit-timing` protocols were designed around the assumption that compositors maintain per-surface queues of content updates. These protocols were implemented in compositors on top of their internal queue-based architectures, and added to `wayland-protocols`. But the core protocol specification was never updated. It still described the old "apply or cache into parent state" model that has no notion of content

updates, and per-surface queues. We now had a situation where the core protocol described one model, extension protocols assumed a different model, and compositors implemented something that sort of bridged both. Implementation and Theory That situation is not ideal: If the internal implementation follows the design which the core protocol implies, you can't deal properly with pending client GPU work, and you can't properly implement the latest timing protocols. To understand and implement the per-surface queue model, you would have to read a whole bunch of discussions, and most likely an implementation such as the one in Mutter. The implementations in compositors also evolved organically, making them more complex than they actually have to be. To make matter worse, we also lacked a shared vocabulary for discussing the behavior. The obvious solution to this is specifying a general model of the per-surface content update queues in the core protocol. Easier said than done though. Coming up with a model that is sufficient to describe the new behavior while also being compatible with the old behavior when no constraints on content updates defer their application was harder than I expected. Together with Julian Orth, we managed to change the Wayland core protocol, and I wrote documentation about the system. Recently Pekka Paalanen and Julian Orth reviewed the work, which allowed it to land. The updated and improved Wayland book should get deployed soon, as well. The end result is that if you ever have to write a Wayland compositor, one of the trickier parts to get right should now be almost trivial. Implement the rules as specified, and things should just work. Edge cases are handled by the general rules rather than requiring special knowledge.

- [Harry Wentland: Plane Color Pipeline, CSC, 3D LUT, and KWin](#) (2026/03/10 00:00)

A wild blog appears... The Plane Color Pipeline API and KWin A couple months ago the DRM/KMS Plane Color Pipeline API was merged after more than 2 years of work and deep discussions. Many people worked on it and it's nice to see it upstream. KWin and other compositors implemented support for it. I'll mainly focus on kwin here because that's what I use regularly and what I am most familiar with. I will also focus on AMD HW because that's what I'm working on. On AMD HW with a kernel that includes the new Color Pipeline API KWin enables HW composition for surfaces that update more than 20 times per second on a single enabled display. It needs a few other things to match as well. In particular, this means that running mpv with the default backend (`--vo=gpu`) will use HW composition for mpv's video surface with the rest of the desktop. The easiest way to observe this is with UMR by running it in `--gui` mode and looking at the KMS tab. In my examples UMR also gets HW composed, so this shows 4 planes: 1920x1200 desktop plane - XR24 720p video plane - AB48 720p UMR plane - XR24 256x256 cursor plane - AR24 The mpv framebuffer shows up as an AB48 buffer, not NV12. This is because the `--vo=gpu` backend in mpv performs any required color-space conversation, scaling, and tone-mapping, and then offers up a 16-bpc buffer to the Wayland compositor, which kwin passes to the DRM/KMS driver. NV12/P010 Scanout We can tell mpv to pass the raw YUV buffer (NV12 or P010) to kwin by passing using the `--vo=dmauf-wayland` backend. This tells mpv to simply decode the video stream but leave the buffer alone. It then passes the buffer information to kwin via the Wayland color-management and color-representation protocol extensions. When we do this we don't see a HW-composed plane in umr. KWin color-space converts, scales, tone-maps, and composes the plane via OpenGL. It can't offload it to display HW because the DRM color pipeline API doesn't yet support color-space conversion (CSC). The `drm_plane` does have `COLOR_RANGE` and `COLOR_ENCODING` properties to specify CSC, but they are deprecated with the color pipeline API. So I went and implemented a CSC `drm_colorop`, added IGT tests and added support for it in kwin. Kernel Patches IGT Patches KWin Patches With this new CSC colorop we now see an NV12 buffer for our SDR video (I'm using a 1080p60 Big Buck Bunny clip). Banding and 3DLUTs Unfortunately we see some banding during the HW composed Big Buck Bunny playback: This is the SW composed version, showing no problems: I haven't yet debugged the banding. It seems to happen with one of the 1D LUTs. But the AMD HW also has a 3D LUT and kwin lets us sample its entire internal color pipeline, so we can simply sample it at our 3DLUT coordinates and program it to

HW. This allows us to represent any complex color pipeline with a single 3D LUT operation. The result is this. kwin branch HDR Video In order to compose HDR content kwin creates a tone-mapper. By packing the entire color pipeline into a 3D LUT we don't need to worry about it and get support for HW composition of HDR content for free. Note: AMD's 3D LUT uses 17 entries per dimension and interpolates tetrahedrally. This will give good results when applied in non-linear luminance space. KWin blends in non-linear space, and the input buffer is non-linear, so it works well here. While this gives good results you might still observe minor differences, especially in brighter areas of the image. This can be observed when toggling between HW and SW composition in certain scenes. Scaling Because AMD's DCN HW uses a multi-tap scaler filter but kwin's SW composition uses GL_LINEAR there are differences in scaling. It's apparent when doing 4-to-1 downscaling, such as when scaling this 4k (HDR) video down to 720p. SW composed: HW composed: The former has stronger aliasing. The latter looks softer and more natural, in my opinion. The difference becomes much less pronounced when the image is downscaled less, e.g., from 4k to 1080p. At this point there is no good API to help align the two. GL seems to only have GL_NEAREST and GL_LINEAR when not dealing with mipmaps. DRM/KMS provides "Default" and "Nearest Neighbor" via the SCALING_FILTER property. While this allows us to use a nearest neighbor filter in both cases that's undesirable since it'll make the image worse in both cases. Seeing is believing When I started this work I asked myself: How do I see whether a surface is a candidate for offloading? How do I see which actual surface is being offloaded? To solve this I worked with Claude to create a plugin that marks surfaces and their offload status: It's quite useful to see immediately which surfaces are candidates, and why they might fail to offload. I've also added a new tab to dynamically toggle HW composition and and off. The screenshot shows toggles for 3D LUT and tone-mapping and while we can add those as well they don't always take effect as expected, so I left them out of the branch that's linked below. But the ability to toggle HW composition is quite powerful when debugging HW composition issues. kwin branch kwin branch on top of csc-3dlut branch UMR DCN tab DCN HW programming can be logged via the amdgpu_dm_dtn_log debug log debugfs. But that log is quite extensive. It can be useful to show this in UMR with auto-update functionality to see programmed settings immediately. The code still needs a fair bit of work as this was the first time I used Claude for something extensive like this. I plan to post it eventually. A Brief Note on LLMs I used Claude Sonnet extensively for this work (it basically wrote all the code), so I thought it prudent to leave a couple thoughts on it. LLMs are large language models, not actual artificial intelligence. They're language models, and are designed to work well with language. They're large and can hold much more context than humans. Use them in ways that uses their strength. I've found value in understanding complex code-bases, and creating code that fits within those code-bases. Don't stop owning your code. Even if it's produced by an LLM, take pride and ownership. This means, review what you get from an LLM. Be active in steering it. Don't throw trash at maintainers. Your name and reputation are on the line. Next Steps I'll be working with relevant communities and maintainers to attempt to upstream these things. The CSC colorop is probably in a good shape. The KWin code requires feedback from maintainers. I expect it will need more work. This also needs more testing. At times I see the 3D LUT fail to apply. I'm not sure whether this is a problem with my kwin code or amdgpu. I don't see offload candidate surfaces from many applications where I'd expect to see them. This needs further analysis. For one, I'm unsure what happens with games. The other thing is Youtube in Firefox, which fails to present the video as an offload surface. Some other videos work fine in Firefox, in particular local video playback. sneaky edit: and power measurements, of course, since that's the entire reason for this.

- [Simon Ser: Status update, February 2026](#) (2026/02/21 22:00)

Hi all! Lars has contributed an implementation independent test suite for the scfg configuration file format. This is quite nice for implementors, they get a base test suite for free. I've added support for it for libscfg, the C implementation. I've spent some time working on the go-proxyproto

library. While adding support for PP2_SUBTYPE_SSL_CLIENT_CERT (a PROXY protocol addition to carry the TLS client certificate I've introduced last month), I've fixed large PROXY protocol headers being rejected (TLS certificates can be a few kilobytes), I've fixed some issues in the test suite, and I've improved the HTTP/2 helper. I've merged support for PP2_SUBTYPE_SSL_CLIENT_CERT in tlstunnel, soju and kimchi. Speaking about soju, delthas and taiite have finished up soju.im/client-cert, a new IRC extension to manage TLS client certificates. Clients can register, unregister, and list TLS client certificates which can be used for authentication for the logged in user. We aim to stop storing plaintext passwords, instead generating a fresh TLS certificate when logging in for the first time and storing its key. Nobody has started working on a Goguma patch yet, but that would be nice! Goguma now has a brand new shiny website! Many thanks to Jean THOMAS for building it from the ground up. delthas has added a /invite command, I've added support for removing reactions (via the new unreact message tag), and I've experimented with a Web build of the app (just for fun, with WebSockets connections instead of TCP). kanshi v1.9 has been released. This new version is the first to leverage vali for Varlink support. The new ...output directive can match any number of outputs, a the new mode preferred output directive can be used to select the mode marked as preferred by the kernel. I've resumed work on oembed-proxy, a small server which generates oEmbed previews for arbitrary URLs. It's quite simple: send an HTTP request with a URL, it replies with a JSON payload with metadata such as page title, image size, and so on. I plan to use it for IRC clients, to show link previews without leaking the client's IP address and to make them work on Web clients. I've added support for Open Graph, the most widely used scheme to attach structured data to Web pages. I ended up linking with ffmpeg because I figured I would need to eventually generate thumbnails for images and videos. I played a bit with CGo to integrate Go's streaming io.Reader with ffmpeg's C API. I had to jump through a few hoops, but it works! Hiroaki Yamamoto has contributed wlroots support for ext-workspace-v1, and Félix Poisot has upgraded color-management-v1 to minor version 2. Félix also uncovered some holes in our explicit synchronization implementation — we're in the process of fixing these up now. I've started the wlroots release candidate cycle, and I just published RC3 today. I've spent quite some time improving go-kdfs, a Go library for the Khronos Data Format Specification. KDFS defines a standard file format to describe how pixels are laid out in memory and how their contents should be interpreted. I've added a bunch of new pixel formats, JSON output for the CLI, unit tests against dfdutils, and a lot of other smaller improvements. I've written a wlroots patch to remove a bunch of manually written pixel format tables and replace them with auto-generated tables from go-kdfs. I've also added sample position to pixfmtdb, a Web frontend for go-kdfs (see for instance the Y samples on the DRM_FORMAT_NV12 page). Next up, I'd like to add missing features to the kdfs compat command so that wlroots can get rid of all of its tables (better endianness support, and flags to specify/strip some information such as the alpha channel, color primaries or transfer function). I'm quite happy with all of the good stuff we've managed to get over the fence this month! See you in March!

- [Christian Gmeiner: GLES3 on etnaviv: Fixing the Hard Parts](#) (2026/02/20 00:00)

This is the start of a series about getting OpenGL ES 3.0 conformance on Vivante GC7000 hardware using the open-source etnaviv driver in Mesa. Thanks to Igalia for giving me the opportunity to spend some time on these topics. Where We Are etnaviv has supported GLES2 on Vivante GPUs for a long time. GLES3 support has been progressing steadily, but the remaining dEQP failures are the stubborn ones - the cases where the hardware doesn't quite do what the spec says, and the driver has to get creative.

- [Dave Airlie \(blogspot\): drm subsystem AI patch review](#) (2026/02/13 06:56)

This topic came up at kernel maintainers summit and some other groups have been playing around with it, particularly the BPF folks, and Chris Mason's work on kernel review prompts[1] for regressions. Red Hat have asked engineers to investigate some workflow enhancements with AI

tooling, so I decided to let the vibecoding off the leash. My main goal:- Provide AI led patch review for drm patches- Don't pollute the mailing list with them at least initially. This led me to wanting to use lei/b4 tools, and public-inbox. If I could push the patches with message-ids and the review reply to a public-inbox I could just publish that and point people at it, and they could consume it using lei into their favorite mbox or browse it on the web. I got claude to run with this idea, and it produced a project [2] that I've been refining for a couple of days. I started with trying to use Chris' prompts, but screwed that up a bit due to sandboxing, but then I started iterating on using them and diverged. The prompts are very directed at regression testing and single patch review, the patches get applied one-by-one to the tree, and the top patch gets the exhaustive regression testing. I realised I probably can't afford this, but it's also not exactly what I want. I wanted a review of the overall series, but also a deeper per-patch review. I didn't really want to have to apply them to a tree, as drm patches are often difficult to figure out the base tree for them. I did want to give claude access to a drm-next tree so it could try apply patches, and if it worked it might increase the review, but if not it would fallback to just using the tree as a reference. Some holes claude fell into, claude when run in batch mode has limits on turns it can take (opening patch files and opening kernel files for reference etc), giving it a large context can sometimes not leave it enough space to finish reviews on large patch series. It tried to inline patches into the prompt before I pointed out that would be bad, it tried to use the review instructions and open a lot of drm files, which ran out of turns. In the end I asked it to summarise the review prompts with some drm specific bits, and produce a working prompt. I'm sure there is plenty of tuning left to do with it. Anyways I'm having my local claude run the poll loop every so often and processing new patches from the list. The results end up in the public-inbox[3], thanks to Benjamin Tissoires for setting up the git to public-inbox webhook. I'd like for patch submitters to use this for some initial feedback, but it's also something that you should feel free to ignore, but I think if we find regressions in the reviews and they've been ignored, then I'll start suggesting it stronger. I don't expect reviewers to review it unless they want to. It was also suggested that perhaps I could fold in review replies as they happen into another review, and this might have some value, but I haven't written it yet. If on the initial review of a patch there is replies it will parse them, but won't do it later. [1] <https://github.com/masoncl/review-prompts>[2] <https://gitlab.freedesktop.org/airlied/patch-reviewer>[3] <https://lore.gitlab.freedesktop.org/drm-ai-reviews/>

- [Christian Gmeiner: My first Vulkan extension](#) (2026/02/13 00:00)

After years of working on etnaviv - a Gallium/OpenGL driver for Vivante GPUs - I've been wanting to get into Vulkan. As part of my work at Igalia, the goal was to bring VK_EXT_blend_operation_advanced to lavapipeline. But rather than going straight there, I started with Honeykrisp - the Vulkan driver for Apple Silicon - as a first target: a real hardware driver to validate the implementation against before wiring it up in a software renderer. My first Vulkan extension, and my first real contribution to Honeykrisp.

- [Adam Jackson: now you're footgunning with gas!](#) (2026/02/10 17:06)

If you haven't heard of gastown it is my sincere pleasure to be the one to fix that. If you're like me and you have way more ideas than time or ability to type them, gastown is an absolute game changer. I haven't felt this jazzed about programming in decades, like, I'm using antiquated slang like "jazzed" without embarrassment. Well. Not embarrassment about that. I awoke to a polite note from a colleague saying I had apparently pushed a bunch of random branches to the upstream Mesa repo and are we sure I wasn't hacked. No, I wasn't, those branches were definitely things I was working on, but I had been working on them locally, nothing should have even been pushed to my personal gitlab repo. I was using gastown to do that work so obviously we start looking there...Gastown is built on beads and beads is built on git. Every atom of work within a gastown has a bead, which means updates to those beads are absolutely critical, if they don't happen then the town chugs to a halt.

Gastown is also built on claude, or whatever, but claude is what I was using. Not to pick on claude, here, just using it as a generic brand name, claude is too polite to be a robust tool, sometimes. It'll lose some critical bit of context and want to stop and ask for directions. For a coding assistant that's a great way to be; for a code factory it's less awesome. It is slightly comic to read how much of gastown is just different ways of exhorting a lilypond of claudes to please do their work, please. In gastown you wrap an upstream project in a rig, and I'm an old so I have all my git clones of everything already in ~/git, so I had set up my rigs to use those as the local storage so I didn't have to wait for things to clone again. The town mayor or one of his underlings dispatches work to ephemeral claude instances by writing the orders down in a bead, the work happens in the worker instance's git worktree instance of the rig. I would use gastown for the automagic git worktree management alone, forget the rest of the automation. But here's where the projectile weapon starts pointing towards the positive gradient of the ol' G field. One way to try to help when claude loses context is to put important orientation information into CLAUDE.md, and helpfully, the /init command will build that for you by inspecting the current project. So I ran that at the top level of my gastown so it, gastown, wouldn't have to keep parsing "gt --help" output just to rediscover how to update a bead, hopefully. In doing so, claude lifted that directive about bead updates being seriously no-kidding mandatory up to the top level. From that point on, claude instances would interpret that directive to apply to the project in the rig! So now, work isn't done until the work-in-progress branch is pushed. Until it is pushed to origin. Which, too bad that you cloned it from https, I'm going to discover the ssh URL to your personal mesa repo from the local repo's git config and change the URL in the rig to use ssh, because I was told to resolve the push failure or else. So, cautionary tale, right? Maybe determinism in your tools still has value. Maybe plaintext English isn't the best idea for a configuration language. Maybe agent prompts need to be extra careful about context. Maybe careful sandbox construction would mitigate that kind of escape. Maybe an open source agent would be more trustworthy in terms of configurable tool usage since you would actually be able to see and control the boundary instead of just trusting that there is a boundary at all.

- [Dave Airlie \(blogspot\): nouveau: a tale of two bugs](#) (2026/02/04 21:04)

Just to keep up some blogging content, I'll do where did I spend/waste time last couple of weeks. I was working on two nouveau kernel bugs in parallel (in between whatever else I was doing). Bug 1: Lyude, 2 or 3 weeks ago identified the RTX6000 Ada GPU wasn't resuming from suspend. I plugged in my one and indeed it wasn't. Turned out since we moved to 570 firmware, this has been broken. We started digging down various holes on what changed, sent NVIDIA debug traces to decode for us. NVIDIA identified that suspend was actually failing but the result wasn't getting propagated up. At least the opengpu driver was working properly. I started writing patches for all the various differences between nouveau and opengpu in terms of what we send to the firmware, but none of them were making a difference. I took a tangent, and decided to try and drop the latest 570.207 firmware into place instead of 570.144. NVIDIA have made attempts to keep the firmware in one stream more ABI stable. 570.207 failed to suspend, but for a different reason. It turns out GSP RPC messages have two levels of sequence numbering, one on the command queue, and one on the RPC. We weren't filling in the RPC one, and somewhere in the later 570's someone found a reason to care. Now it turned out whenever we boot on 570 firmware we get a bunch of async msgs from GSP, with the word ASSERT in them with no additional info. Looks like at least some of those messages were due to our missing sequence numbers and fixing that stopped those. And then? still didn't suspend/resume. Dug into memory allocations, framebuffer suspend/resume allocations. Until Milos on discord said you did confirm the INTERNAL_FBSR_INIT packet is the same, and indeed it wasn't. There is a flag bEnteringGCOff, which you set if you are entering into graphics off suspend state, however for normal suspend/resume instead of runtime suspend/resume, we shouldn't tell the firmware we are going to gcoff for some reason. Fixing that fixed suspend/resume. While I was head down on fixing this, the bug trickled up into a few other places and I had

complaints from a laptop vendor and RH internal QA all lined up when I found the fix. The fix is now in `drm-misc-fixes`. Bug 2: A while ago Mary, a nouveau developer, enabled larger pages support in the kernel/mesa for nouveau/nvk. This enables a number of cool things like compression and gives good speedups for games. However Mel, another nvk developer reported random page faults running Vulkan CTS with large pages enabled. Mary produced a workaround which would have violated some locking rules, but showed that there was some race in the page table reference counting. NVIDIA GPUs post pascal, have a concept of a dual page table. At the 64k level you can have two tables, one with 64K entries, and one with 4K entries, and the addresses of both are put in the page directory. The hardware then uses the state of entries in the 64k pages to decide what to do with the 4k entries. nouveau creates these 4k/64k tables dynamically and reference counts them. However the nouveau code was written pre VMBIND, and fully expected the operation ordering to be reference/map/unmap/unreference, and we would always do a complete cycle on 4k before moving to 64k and vice versa. However VMBIND means we delay unrefs to a safe place, which might be after refs happen. Fun things like ref 4k, map 4k, unmap 4k, ref 64k, map 64k, unref 4k, unmap 64k, unref 64k can happen, and the code just wasn't ready to handle those. Unref on 4k would sometimes overwrite the entry in the 64k table to invalid, even when it was valid. This took a lot of thought and 5 or 6 iterations on ideas before we stopped seeing fails. In the end the main things were to reference count the 4k/64k ref/unref separately, but also the last thing to do a map operation owned the 64k entry, which should conform to how userspace uses this interface. The fixes for this are now in `drm-misc-next-fixes`. Thanks to everyone who helped, Lyude/Milos on the suspend/resume, Mary/Mel on the page tables.

- [Natalie Vock: Inside Mesa 26.0's RADV RT improvements](#) (2026/01/30 00:00)

Mesa 26.0 is big for RADV's ray tracing. In fact, it's so big it single-handedly revived this blog. There are a lot of improvements to talk about, and some of them were in the making for a little over two years at this point. In this blog post I'll focus on the things I myself worked on specifically, most of which revolve around how ray tracing pipelines are compiled and dispatched. Of course, there's more than just what I did myself: Konstantin Seurer worked on a lot of very cool improvements to how we build BVHs, the data structure that RT hardware uses for the triangle soup making up the geometry in game scenes so the HW can trace rays against them efficiently. RT pipeline compilation The rest of this blog post will assume some basic idea of how GPU ray tracing and ray tracing pipelines work. I've written about this in more detail one-and-a-half years ago, in my blog post about RT pipeline being enabled by default. Let's take a bit of a closer look at what I said about RT pipelines in RADV back then. In a footnote, I said: Any-hit and Intersection shaders are still combined into a single traversal shader. This still shows some of the disadvantages of the combined shader method, but generally compile times aren't that ludicrous anymore. I spent a significant amount of time in that blogpost detailing about how there tend to be a really large number of shaders, and combining them into a single megashader is very slow because shader sizes get genuinely ridiculous at that point. So clearly, it was only a matter of time until the any-hit/intersection shader combination would blow up spectacularly on a spectacular number of shaders, as well. So there's this thing called Unreal Engine For illustrating the issues with inlined any-hit/intersection shaders, I'll use Unreal Engine as an example because I noticed it being particularly egregious here. This definitely was an issue with other RT games/workloads as well, and function calls will provide improvements there too. There's a lot of people going around making fun of Unreal Engine these days, to the point of entire social media presences being built around mocking the ways in which UE is inefficient, slow, badly-designed bloatware and whatnot. Unfortunately, the most popular critics often know the least what they're actually talking about. I feel compelled to point out here that while there certainly are reasonable complaints to be raised about UE and games made with it, I explicitly don't want this section (or anything else in this post, really) to be misconstrued as "UE doing a bad thing". As you'll see, Unreal is really just using the RT pipeline API as designed. With the disclaimer aside, what does Unreal actually do here that made RADV fall over

so hard? Let's talk a bit about how big game engines handle shading and materials. As you'll probably know already, to calculate how lighting interacts with objects in a scene, an application will usually run small programs called "shaders" on the GPU that, among other things¹, calculate the colors different pixels have according to the material at that pixel. Different materials interact with light differently, and in a large world with tons of different materials, you might end up having a ton of different shaders. In a traditional raster setup you draw each object separately, so you can compile a lot of graphics pipelines for all of your materials, and then bind the correct one whenever you draw something with that material. However, this approach falls apart in ray tracing. Rays can shoot through the scene randomly and they can hit pretty much any object that's loaded in at the moment. You can only ever use one ray tracing pipeline at once, so every single material that exists in your scene and may be hit by a ray needs to be present in the RT pipeline. The more materials a game has, the more ludicrous the number of shaders gets. Usually, this is most relevant for closest-hit shaders, because these are the shaders that get called for the object hit by the ray (where shading needs to be calculated). However, depending on your material setup, you may have something like translucent materials - where parts of the material are "see-through", and rays should go through these parts to reveal the scene behind it instead of stopping. This is where any-hit shaders come into play - any-hit shaders can instruct the driver to ignore a ray hitting a geometry, and instead keep searching for the next hit. If you have a ton of (potentially) translucent materials, that would translate into a lot of any-hit shaders being compiled for these materials. The design of RT pipelines is quite obviously written in a way that accounts for this. In the previous blogpost I already mentioned pipeline libraries - the idea is that a material could just be contained in a "library", and if RT pipelines want to use it, they just need to link to the library instead of compiling the shading code all over again. This also allows for easy addition/removal of materials: Even though you have to re-create the RT pipeline, all you need to do is link to the already compiled libraries for the different materials. UE, particularly UE4, is a heavy user of libraries, which makes a lot of sense: It maps very well to what it's trying to achieve. Everything's good, as long as the driver doesn't do silly things. Silly things like, for example, combining any-hit shaders into one big traversal shader. Doing something like that pretty much entirely side-steps the point of libraries. The traversal shader can only be compiled when all any-hit shaders are known, which is only at the very final linking step, which is supposed to be very fast... And if UE4, assuming the linking step is very fast, does that re-linking over and over, very often, what you end up with is horrible pipeline compilation stutter every few seconds. And in this case, it's not really UE's fault, even! Sorry for that, Unreal. Why can't we just compile any-hit/intersection separately? Clearly, inlining all the any-hit and intersection shaders won't work. So why not just compile them separately? To answer that, I'll try to start with explaining some assumptions that lie at the base of RADV's shader compilation. When ACO (and NIR, too) were written, shaders were usually incredibly simple. They had some control flow, ifs, loops and whatnot, but all the code that would ever execute was contained in one compact program executing top-to-bottom. This perfectly matched what graphics/compute shaders looked like in the APIs, and what the API does is what you want to optimize for. Unfortunately, this means RADV's shader compilation stack got hit extra hard by the paradigm shift introduced by RT pipelines. Dynamic linking of different programs, and calls across the dynamic link boundaries, is something common in CPU programming languages (C/C++, etc.), but Mesa never really had to deal with something like that before². One specific core assumption that prevents us from compiling any-hit/intersection shaders separately just like that is that every piece of code assumes it has exclusive and complete access to things like registers and other hardware resources. Comparing to CPU again, most of the program code is contained in some functions, and those functions will be called from somewhere else³. Those functions will have used CPU registers and stack memory and so on before, and code inside that function can't write to just any CPU register, or any location on stack. Which registers are writable by a function and which ones must have their values preserved (so that the function callers can store values of their own

there without them being overwritten) are governed by little specifications called “calling conventions”. In Mesa, the shader compiler generally used to have no concept of calling conventions, or a concept of “calling” something, for that matter. There was no concept of a register having some value from a function caller and needing to be preserved - if a register exists, the shader might end up writing its own value to it. In cases of graphics/compute shaders, this wasn’t a problem - the registers only ever had random uninitialized values in them. This has always been a problem for separately compiling shaders in RT pipelines, but we had a different solution: At every point a shader called another shader, we’d split the shader in half: One half containing everything before the call, and the other half containing everything after. Of course, sometimes the second half needed variables coming from the first half of the shader. All these variables would be stored to memory in the first half. Then, the first half ends, and execution jumps to the called shader. Once the end of the called shader is reached, execution returns to the second half. This was good enough for things like calling into traceRay to trace a ray and execute all the associated closest hit/miss shaders. Usually, applications wouldn’t have that many variables needing to be backed up to memory, and tracing a ray is supposed to be expensive. But that concept completely breaks down when you apply it to any-hit shaders. At the point an any-hit shader is called, you’re right in the middle of ray traversal. Ray traversal has lots of internal state variables that you really want to keep in registers at all times. If you call an any-hit shader with this approach, you’d have to back up all of these state variables to memory and reload them back afterwards. Any-hit shaders are supposed to be relatively cheap and called potentially lots of times during traversal. All these memory stores and reloads you’d need to insert would completely ruin performance. So, separately compiling any-shaders was an absolute no-go. At least, unless someone were to go off the deep end and change the entire compiler stack to fix the assumptions at their heart. “So, where have you been the last two years?” I went and changed more or less the entire compiler stack to fix these assumptions and introduce proper function calls. The biggest part of this work by far were the absolute basics. How do we best teach the compiler that certain registers need to be preserved and are best left alone? How should the compiler figure out that something like a call instruction might randomly overwrite other registers? How do we represent a calling convention/ABI specification in the driver? All of these problems can be tackled with different approaches and at different stages of compilation, and nailing down a clean solution is pretty important in a rework as fundamental as this one. I started out with applying function calls to the shaders that were already separately compiled - this means that the function call work itself didn’t improve performance by too much, but in retrospect I think it was a very good idea to make sure the baseline functionality is rock-solid before moving on to separately-compiling any-hit shaders. Indeed, once I finally got around to adding the code that splits out any-hit/intersection shaders and use function calls for them, things worked nearly out of the box! I opened the associated merge request a bit over two weeks ago and got everything merged within a week. (Of course, I would never have gotten it in that fast without all the reviewers teaming up to get everything in ASAP! Big thank you to Daniel, Rhys and Konstantin) In comparison, I started work on function calls in January of 2024 and got the initial code in a good enough shape to open a merge request in June that year, and the code only got merged on the same day I opened the above merge request, two years after starting the initial drafting (although to be fair, that merge request also had periods of being stalled due to personal reasons). Shader compilation with function calls Function calls makes shader compilation work in arguably a much more straightforward way. For the most part, the shader just gets compiled like any other - there’s no fancy splitting or anything going on. If a shader calls another shader, like when executing traceRay, or when calling an any-hit shaders, a call instruction is generated. When the called shader finishes, execution resumes after the call instruction. All the magic happens in ACO, the compiler backend. I’ve documented the more technical design of how calls and ABIs are represented in a docs article. At first, call instructions in the NIR IR are translated to a p_call “pseudo” instruction. It’s not actually a hardware instruction, but serves as a placeholder for the eventual

jump to the callee. This instruction also carries information about which specific registers parameters will be stored in, and which registers may be overwritten by the call instruction. ACO's compiler passes have special handling for calls wherever necessary: For example, passes analyzing how many registers are required in all the different parts of the code take special care to take into account that in call instructions, fewer registers may be available to store values in (because all other values are overwritten). ACO also has a spilling pass for moving register values to memory whenever the amount of used registers exceeds the available amount. Another fundamental change is that function calls also introduce a call stack. In CPUs, this is no big deal - you have one stack pointer register, and it points to the stack region that your program uses. However, on GPUs, there isn't just one stack - remember that GPUs are highly parallel, and every thread running on the GPU needs its own stack! Luckily, this sounds worse at first than it actually is. In fact, the hardware already has facilities to help manage stacks. AMD GPUs ever since Vega4 have the concept of "scratch memory" - a memory pool in VRAM where the hardware ensures that each thread has its own private "scratch region". There are special `scratch_*` memory instructions that load and store from this scratch area. Even though they're also VRAM loads/stores, they don't take any address, just an offset, and for each thread return the value stored in that thread's own scratch memory region. In my blog post about RT pipeline being enabled by default I claimed AMD GPUs don't implement a call stack. This is actually misleading - the scratch memory functionality is all you need to implement a stack yourself. The "stack pointer" here is just the offset you pass to the `scratch_*` memory instruction. Pushing to the stack increases the stack offset, and popping from it decreases the offset⁵. Eventually, when it comes to converting a call to hardware instructions, all that is needed is to execute the `s_swappc` instruction. This instruction automatically writes the address of the next instruction to a register before jumping to the called shader. When the called shader wants to return, it merely needs to jump to the address stored in that register, and execution resumes from right after the call instruction. Finally, any-hit separate compilation was a straightforward task as well - it was merely an issue of defining an ABI that made sure that a ton of registers stay preserved and the caller can stash its values there. In practice, all of the traversal state will be stashed in these preserved registers. No expensive spilling to memory needed, just a quick jump to the any-hit shader and back. Performance considerations If you look at the merge request, the performance benefits seem pretty obvious. Ghostwire Tokyo's RT passes speed up by more than 2x, and of course pipeline compilation times improved massively. The compilation time difference is quite easy to explain. Generally, compilers will perform a ton of analysis passes on shader code to find everything they can to optimize it to death. However, these analysis passes often require going over the same code more than once, e.g. after gathering more context elsewhere in the shader. This also means that a shader that doubles in size will take more than twice as long to compile. When inlining hundreds or thousands of shaders into one, that also means that shader's compile time grows by a lot more than just a hundred or a thousand times. Thus, if we reverse things and are suddenly able to stop inlining all the shaders into one, that scaling effect means all the shaders will take less total time to compile than the one big megashader. In practice, all modern games also offload shader compilation to multiple threads. If you can compile the any-hit shaders separately, the game can compile them all in parallel - this just isn't possible with the single megashader which will always be compiled on a single thread. In the runtime performance department, moving to just having a single call instruction instead of hundreds of shaders in one place means the loop has a much smaller code size. In a loop iteration where you don't call any any-hit shaders, you would still need to jump over all of the code for those shaders, almost certainly causing instruction cache misses, stalls and so on. Also, forcing any-hit/intersection shaders to be separate also means that any-hit/intersection shaders that consume tons of registers despite nearly never getting called won't have any negative effects on ray traversal as a whole. ACO has heuristics on where to optimally insert memory stores in case something somewhere needs more registers than available. However, these heuristics may decide to insert memory stores inside the generic

traversal loop, even if the problematic register usage only comes from a few rarely-called inlined shaders. These stores in the generic loop would now mean that the whole shader is slowed down in every case. However, separate compilation doesn't exclusively have advantages, either. In an inlined shader, the compiler is able to use the context surrounding the (now-inlined) shader to optimize the code itself. A separately-compiled shader needs to be able to get called from any imaginable context (as long as it conforms to ABI), and this inhibits optimization. Another consideration is that the jump itself has a small cost (not as big as you'd think, but it does have a cost). RADV currently keeps inlining any-hit shaders as long as you don't have too many of them, and as long as doing so wouldn't inhibit the ability to compile the shaders in parallel. About that big UE5 Lumen perf improvement I also opened a merge request that provided massive performance improvements to Lumen's RT right before the branchpoint. However, these improvements are completely unrelated to function calls. In fact, they're a tiny bit embarrassing, because all that changed was that RADV doesn't make the hardware do ridiculously inefficient things anymore. Let's talk about dispatching RT shaders. The Vulkan API provides a `vkCmdTraceRaysKHR` command that takes in the number of rays to dispatch for X, Y and Z dimensions. Usually, compute dispatches are described in terms of how many thread groups to dispatch, but RT is special because one ray corresponds to one thread. So here, we really get the dispatch sizes in threads, not groups. By itself, that's not an issue. In fact, AMD hardware has always been able to specify dispatch dimensions in threads instead of groups. In that case, the hardware takes the job of assembling just enough groups that hold the specified number of threads. The issue here comes from how we describe that group to the hardware. The workgroup size itself is also per-dimension, and the simplest case of 32x1x1 threads (i.e. a 1D workgroup) is actually not always the best. Let's consider a very common ray tracing use case: You might want to trace a ray for each pixel in a 1920x1080 image. That's pretty easy, you just call `vkCmdTraceRaysKHR` to dispatch 1920 rays in the X dimension and 1080 in the Y dimension. When you dispatch a 32x1x1 workgroup, the coordinates for each thread in a workgroup look like this: thread id | 0 | 1 | 2 | ... | 16 | 17 | ... | 31 | coord |(0,0)|(1,0)|(2,0)| ... |(16,0)|(17,0)|...|(31,0)| Or, if you consider how the thread IDs are laid out in the image: ----- 0 | 1 | 2 | 3 | .. ----- That's a straight line in image space. That's not the best, because it means that the pixels will most likely cover different objects which may have very different trace characteristics. This means divergence during RT will be higher, which can make the overall process slower. Let's look instead what happens when you make the workgroup 2D, with a 8x4 size: thread id | 0 | 1 | 2 | ... | 16 | 17 | ... | 31 | coord |(0,0)|(1,0)|(2,0)| ... |(0,2) |(1,2) |...|(7,3) | In image space: ----- 0 | 1 | 2 | 3 | .. ----- 8 | 9 | 10| 11| .. ----- 16| 17| 18| 19| .. ----- That's much better. Threads are now arranged in a little square, and these squares are much more likely to all cover the same objects, have similar RT characteristics, etc. This is why RADV used 8x4 workgroups as well. Now let's get to when this breaks down. What if the RT dispatch doesn't actually have 2 dimensions? What if there are 1920 rays in the X dimension, but the Y dimension is just 1? It turns out that the hardware can only run 8 threads in a single wavefront in this case. This is because the rest of the workgroup is out-of-bounds of the dispatch - it has a non-zero Y coordinate, but the size in the Y dimension is only 1, so it would exceed the dispatch bounds. The hardware also can't pull in threads from other workgroups, because one wavefront can only ever execute one workgroup. The end result is that the wave runs with only 8 out of 32 threads active - at 1/4 theoretical performance. For no real reason. I actually had noticed this issue years ago (with UE4, ironically). Back then I worked around it by rearranging the game's dispatch sizes into a 2D one behind its back, and recalculating a 1-dimensional dispatch ID inside the RT shader so the game doesn't notice. That worked just fine... as long as we're actually aware about the dispatch sizes. UE5 doesn't actually use `vkCmdTraceRaysKHR`. It uses `vkCmdTraceRaysIndirectKHR`, a variant of the command where the dispatch size is read from GPU memory, not specified on the CPU. This command is really cool and allows for some and nifty GPU-driven rendering setups where you only dispatch as many rays as you're definitely

going to trace (as determined by previous GPU commands). This command also rips a giant hole in the approach of rearranging dispatch sizes, because we don't even know the dispatch size before the dispatch is actually executed. That means the super simple workaround I built was never hit, and we had the same embarrassingly inefficient RT performance as a few years ago all over again. Obviously, if UE5 is too smart for your workaround, then the solution is to make an even smarter workaround. The ideal solution would work with a 1D thread ID (so that we don't run into any more issues when there is a 1D dispatch, but if a 2D dispatch is detected, we turn that "line" of 1D IDs into a "square". The whole idea about turning a linear coordinate into a square reminded me a lot of how Z-order curves work. In fact, the GPU arranges things like image data on a Z-order curve by interleaving the address bits from X and Y already, because nearby pixels are often accessed together and it's better if they're close to each other. However, instead of interleaving a X and Y coordinate pair to make a linear memory address, we want the opposite: We have a linear dispatch ID, and we want to recover a 2D coordinate inside a square from it. That's not too hard, you just do the opposite operation: Deinterleave the bits, where every odd/even bit of the dispatch ID forms the X/Y coordinate. As it turned out, you can actually do this entirely from inside the shader with just a few bit twiddling tricks, so this approach work for both indirect and direct (non-indirect) trace commands. With that approach, dispatch IDs and coordinates look something like this: thread id | 0 | 1 | 2 | ... | 16 | 17 | ... | 31 | coord
 |(0,0)|(1,0)|(0,1)| ... |(4,0) |(5,0) |...|(7,3) | In image space: ----- 0 | 1 | 4 | 5 | .. ----- 2 | 3 | 6 | 7 | .. ----- 8 | 9 | 12| 13| .. -
 ----- 10| 11| 14| 15| .. ----- Not only are the thread IDs now arranged in squares, the squares themselves get recursively subdivided into more squares! I think theoretically this should be a further improvement w.r.t divergence, but I don't think it has resulted in measurable speedup in practice anywhere. The most important thing, though, is that now UE5 RT doesn't run 4x slower than it should. Oops. Bonus content: Function call bug bonanza The second most fun thing about function calls is that you can just jump to literally any program anywhere, provided the program doesn't completely thrash your preserved registers and stack space. The most fun thing about function calls is what happens when the program does just that. I'm going to use this section to scream in the void about two very real function call bugs that were reported after I already merged the MR. This is not an exhaustive list, you can trust I've had much much more fun just like what I'll be presenting here while I was testing and developing function calls. Avowed gets stuck in an infinite loop On the scale of function call bugs, this one was rather tame, even. Having infinite loops isn't the most optimal for hang debugging, but it does mean that you can use a tool like umr to sample which wavefronts are active, and get some register dumps. The program counter will at least point to some instruction in the loop that it's stuck in, and you can get yourself the disassembly of the whole shader to try and figure out what's going in the loop and why the exit conditions aren't met. The loop in Avowed was rather simple: It traced a ray in a loop, and when the loop counter was equal to an exit value, control flow would break out of the loop. The register dumps also immediately highlighted the loop exit counter being random garbage. So far so good. During the traceRay call, the loop exit counter was backed up to the shader's stack. Okay, so it's pretty obvious that the stack got smashed somehow and that corrupted the loop exit counter. What was not obvious, however, was what smashed the stack. Debugging this is generally a bit of an issue - GPUs are far, far away from tools like AddressSanitizer, especially at a compiler level. There are no tools that would help me catch a faulty access at runtime. All I could really do was look at all the shaders in that ray tracing pipeline (luckily that one didn't have too many) and see if they somehow store to wrong stack locations. All shaders in that pipeline were completely fine, though. I checked every single scratch instruction in every shader if the offsets were correct (luckily, the offsets are constants encoded in the disassembly, so this part was trivial). I also verified that the stack pointer was incremented by the correct values - everything was completely fine. No shader was smashing its callers' stack. I found the bug more or less by complete chance. The shader code was indeed completely correct, there were no miscompilations

happening. Instead, the “scratch memory” area the HW allocated was smaller than what each thread actually used, because I forgot to multiply by the number of threads in a wavefront in one place. The stack wasn’t smashed by the called function, it was smashed by a completely different thread. Whether your stack would get smashed was essentially complete luck, depending on where the HW placed your scratch memory area, other wavefront’s scratch, and how those wavefronts’ execution was timed relative to yours. I don’t think I would ever have been able to deduce this from any debugger output, so I should probably count myself lucky I stumbled upon the fix regardless. Silent Hill 2’s reflections sample the sky color Did I talk about Unreal Engine yet? Let’s talk about Unreal Engine some more. Silent Hill 2 uses Lumen for its reflection/GI system, and somehow Lumen from UE 5.3 specifically was the only thing that seemed to reproduce this particular bug. In every way the Avowed bug was tolerable to debug, this one was pure suffering. There were no GPU hangs, all shaders ran completely fine. That means using umr and getting a rough idea of where the issue is was off the table from the start. Unfortunately, the RT pipeline was also way too large to analyze - there were a few hundred hit shaders, but there also were seven completely different ray generation shaders. Having little other recourse, I started trying to at least narrow down the ray generation shader that triggered the fault. I used Mesa’s debugging environment variables to dump the SPIR-V of all the shaders the driver encountered, and then used spirv-cross on all of them to turn them into editable GLSL. For each ray generation shader, I’d comment out the imageStore instructions that stored the RT result to some image, recompiled the modified GLSL to SPIR-V, and instructed Mesa to sneakily swap out the original ray-gen SPIR-V with my modified one. Then I re-ran the game to see if anything changed. This indeed led me to find the correct ray generation shader, but the lead turned into a dead end - there was little insight other than that the ray was indeed executing the miss shader. Everything seemed correct so far, and if I hadn’t known these rays didn’t miss about 3 commits ago, I honestly wouldn’t even have suspected anything was wrong at all. The next thing I tried was commenting out random things in ray traversal code. Skipping over all any-hit/intersection shaders yielded no change, and neither did replacing the ray flags/culling masks with known good constants to rule out wrong values being passed as parameters. What did “fix” the result, however, was... commenting out the calls to closest-hit shaders. Now, if closest-hit shaders get called and that makes miss shaders execute somehow, you’d perhaps think we’d be calling the wrong function. Maybe we confuse the shader binding table where we load the addresses of shaders to call from? To verify that assumption, I also disabled calling any and all miss shaders. I zeroed out the addresses in the shader handles to make extra sure there was no possible way that a miss shader could ever get called. To keep things working, I replaced the code that calls miss shaders with the relevant code fragment from UE’s miss shader (essentially inlining the shader myself). Nothing changed from that. That means a closest-hit shader being executed somehow resulted in a ray traversal itself returning a miss, not the wrong function being called. Perhaps the closest-hit shaders corrupt some caller values again? Since the RT pipeline was too big to analyze, I tried to narrow down the suspicious shaders by only disabling specific closest-hit shaders. I also discovered that just making all closest-hit shaders no-ops “fixed” things as well, even if they do get called. Sure enough, at some point I had a specific closest-hit shader where the issue went away once I deleted all code from it/made it a no-op. I even figured out a specific register that, if explicitly preserved, would make the issue go away. The only problem was that this register corresponded to one part of the return value of the closest-hit shader - that is, a register that the shader was supposed to overwrite. From here on out it gets completely nonsensical. I will save you the multiple days of confusion, hair-pulling, desperation and agony over the complete and utter undebuggableness of Lumen’s RT setup and skip to the solution: It turned out the “faulty” closest-hit shader I found was nothing but a red herring. Lumen’s RT consists of 6+ RT dispatches, most of which I haven’t exactly figured out the purpose of, but what I seemed to observe was that the faulty RT dispatch used the results of the previous RT dispatch to make decisions on whether to trace any rays or not. Making the closest-hit shaders a no-op did nothing but disable the subsequent traceRays

that actually exhibited the issue. Since these RT dispatches used the same RT pipelines, that meant virtually any avenue I had of debugging this driver-side was completely meaningless. Any hacks inside the shader compiler might actually work around the issue, or just affect a conceptually unrelated dispatch that happens to disable the actually problematic rays. Determining which was the case was nearly impossible, especially in a general case. I never really figured out how to debug this issue. Once again, what saved me was a random epiphany out of the blue. In fact, now that I know what the bug was, I'm convinced I would've never found this through a debugger either. The issue turned out to be in an optimization for what's commonly called tail-calls. If you have a function that calls another function at the very end just before returning, a common optimization is to simply turn that call into a jump, and let the other function return directly to the caller. Imagine ray traversal working a bit like this C code: `/* hitT is the t value of the ray at the hit point */ payload closestHit(float hitT); /* tMax is the maximum range of the ray, if there is * no hit with a t <= tMax, the ray misses instead */ payload traversal(float tMax) { do something; if (hit) return closestHit(hitT); // gets replaced with a jmp, closestHit returns directly to traversal's caller } More specifically, the bug was with how preserved parameters and tail-calls interact. Function callers are generally supposed to assume that preserved parameters do not change their value over the function call. That means it's safe to reuse that register after the call and assuming it still has the value the caller put in. However, in the example above, let's assume closestHit has the same calling convention as traversal. That means closestHit's parameter needs to go into the same register as traversal's parameter, and thus the register gets overwritten. If traversal's caller was assuming that the parameter is preserved, that would mean the value of tMax has just been overwritten with the value of hitT without the caller knowing. If traversal now gets called again from the same place, the value of tMax is not the intended value, but the hitT value from the previous iteration, which is definitely smaller than tMax. Put shortly: If all these conditions are met, a smaller-than-intended tMax could cause rays to miss when they were intended to hit. Once again, I got incredibly lucky and stumbled upon the bug by complete chance. The GPU gods seem to be in good spirits for my endeavours. I pray it stays this way. Footnotes "Shader" in this context really means any program that runs on the GPU. The RT pipeline is also made of shaders, shaders determine where the points and triangles making up each object end up on screen, there are compute shaders for generic computing, and so on... ↵ There actually is another use-case where this becomes relevant on GPU - and that is GPGPU code like CUDA/HIP/OpenCL. CUDA/HIP allow you to write C++ for the GPU in a much more "CPU-like" programming environment (OpenCL uses C), and you run into all the same problems there. This also means all the major GPU vendors had already written their solutions for these problems when raytracing came around. There are OpenCL kernels that end up really really bad if you don't have proper function calls in the compiler (which Rusticl suffers from right now), and the function calls work in RADV/ACO may end up proving useful for those as well. ↵ Even your main function works like that, actually. Unless you have some form of freestanding environment, all your program code works like that. ↵ In RADV, the stack pointer is actually constant across a function, and pushing/popping to/from the stack is implemented by adding another offset to the constant stack pointer in load/store instructions. This allows to make the stack pointer an SGPR instead of a VGPR and simplifies stack accesses that aren't push/pop. ↵ We support raytracing before Vega too. We support function calls on all GPUs, as well, through a little magic in dreaming up a buffer descriptor with specific memory swizzling to achieve the same addressing that scratch_* instructions use on Vega and later. ↵`

- [Lennart Poettering: Introducing Amutable](#) (2026/01/26 23:00)

Today, we announce Amutable, our new company. We - @blixtra@hachyderm.io, @brauner@mastodon.social, @davidstrauss@mastodon.social, @rodrigo_rata@mastodon.social, @michaelvogt@mastodon.social, @pothos@fosstodon.org, @zbyszek@fosstodon.org, @daandemeyer@mastodon.social @cyphar@mastodon.social, @jrocha@floss.social and yours truly - are building the

□ next generation of Linux systems, with integrity, determinism, and verification – every step of the way. For more information see → <https://amutable.com/blog/introducing-amutable>

- [Mike Blumenkrantz: Unpopular Opinion](#) (2026/01/23 00:00)

A Big Day For Graphics Today is a big day for graphics. We got shiny new extensions and a new RM2026 profile, huzzah. `VK_EXT_descriptor_heap` is huge. I mean in terms of surface area, the sheer girth of the spec, and the number of years it's been under development. Seriously, check out that contributor list. Is it the longest ever? I'm not about to do comparisons, but it might be. So this is a big deal, and everyone is out in the streets (I assume to celebrate such a monumental leap forward), and I'm not. All hats off. Person to person, let's talk. Power Overwhelming It's true that descriptor heap is incredibly powerful. It perfectly exemplifies everything that Vulkan is: low-level, verbose, flexible. `vkd3d-proton` will make good use of it (eventually), as this more closely relates to the DX12 mechanics it translates. Game engines will finally have something that allows them to footgun as hard as they deserve. This functionality even maps more closely to certain types of hardware, as described by a great `gfxstrand` blog post. There is, to my knowledge, just about nothing you can't do with `VK_EXT_descriptor_heap`. It's really, really good, and I'm proud of what the Vulkan WG has accomplished here. But I don't like it. What Is This Incredibly Hot Take? It's a risky position; I don't want anyone's takeaway to be "Mike shoots down new descriptor extension as worst idea in history". We're all smart people, and we can comprehend nuance, like the difference between `rb` and `ab` in EGL patch review (protip: if anyone ever gives you an `rb`, they're fucking lying because nobody can fully comprehend that code). In short, I don't expect `zink` to ever move to descriptor heap. If it does, it'll be years from now as a result of taking on some other even more amazing extension which depends on heaps. Why is this, I'm sure you ask. Well, there's a few reasons: Code Complexity Like all things Vulkan, "getting it right" with descriptors meant creating an API so verbose that I could write novels with fewer characters than some of the struct names. Everything is brand new, with no sharing/reuse of any existing code. As anyone who has ever stepped into an unfamiliar bit of code and thought "this is garbage, I should rewrite it all" knows too well, existing code is always the worst code—but it's also the code that works and is tied into all the other existing code. Pretty soon, attempting to parachute in a new descriptor API becomes rewriting literally everything because it's all incompatible. Great for those with time and resources to spare, not so great for everyone else. Gone are image views, which is cool and good, except that everything else in Vulkan still uses them, meaning now all image descriptors need an extra pile of code to initialize the new structs which are used only for heaps. Hope none of that was shared between rendering and descriptor use, because now there will be rendering use and descriptor use and they are completely separate. Do I hate image views? Undoubtedly, and I like this direction, but hit me up in a few more years when I can delete them everywhere. Shader interfaces are going to be the source of most pain. Sure, it's very possible to keep existing shader infrastructure and use the mapping API with its glorious nested structs. But now you have an extra 1000 lines of mapping API structs to juggle on top. Alternatively, you can get AI to rewrite all your shaders to use the new `spirv` extension and have direct heap access. Performance Descriptor heap maps closer to hardware, which should enable users to get more performant execution by eliminating indirection with direct heap access. This is great. Full stop. ...Unless you're like `zink`, where the only way to avoid shredding 47 CPUs every time you change descriptors is to use a "sliding" offset for descriptors and update it each draw (i.e., `VK_DESCRIPTOR_MAPPING_SOURCE_HEAP_WITH_PUSH_INDEX_EXT`). Then you can't use direct heap access. Which means you're still indirecting your descriptor access (which has always been the purported perf pain point of 1.0 descriptors and `EXT_descriptor_buffer`). You do not pass Go, you do not collect \$200. All you do is write a ton of new code. Opinionated Development There's a tremendous piece of exposition outlining the reasons why `EXT_descriptor_heap` exists in the proposal. None of these items are incorrect. I've even contributed to this document. If I were

writing an engine from scratch, I would certainly expect to use heaps for portability reasons (i.e., in theory, it should eventually be available on all hardware). But as flexible and powerful as descriptor heap is, there are some annoying cases where it passes the buck to the user. Specifically, I'm talking about management of the sampler heap. 1.0 descriptors and descriptor buffer just handwave away the exact hardware details, but with `VK_EXT_descriptor_heap`, you are now the captain of your own destiny and also the manager of exactly how the hardware is allocating its samplers. So if you're on NVIDIA, where you have exactly 4096 available samplers as a hardware limit, you now have to juggle that limit yourself instead of letting the driver handle it for you. This also applies to border colors, which has its own note in the proposal. At an objective, high-view level, it's awesome to have such fine-grained control over the hardware. Then again, it's one more thing the driver is no longer managing. I Don't Have A Better Solution That's certainly the takeaway here. I'm not saying go back to 1.0 descriptors. Nobody should do that. I'm not saying stick with descriptor buffers either. Descriptor heap has been under development since before I could legally drive, and I'm certainly not smarter than everyone (or anyone, most likely) who worked on it. Maybe this is the best we'll get. Maybe the future of descriptors really is micromanaging every byte of device memory and material stored within because we haven't read every blog post in existence and don't trust driver developers to make our shit run good. Maybe OpenGL, with its drivers that "just worked" under the hood (with the caveat that you, the developer, can't be an idiot), wasn't what we all wanted. Maybe I was wrong, and we do need like five trillion more blog posts about Vulkan descriptor models. Because releasing a new descriptor extension is definitely how you get more of those blog posts. I'm tired, boss.

- [Simon Ser: Status update, January 2026](#) (2026/01/21 22:00)

Hi! Last week I've released Goguma v0.9! This new version brings a lot of niceties, see the release notes for more details. New since last month are audio previews implemented by delthas, images for users, channels & networks, and usage hints when typing a command. Jean THOMAS has been hard at work to update the iOS port and publish Goguma on AltStore PAL. It's been a while since I've started a NPotM, but this time I have something new to show you: nagjo is a small IRC bot for Forgejo. It posts messages on activity in Forgejo (issue opened, pull request merged, commits pushed, and so on), and it expands references to issues and pull requests in messages (writing "can you look at #42?" will reply with the issue's title and link). It's very similar to glhf, its GitLab counterpart, but the configuration file enables much more flexible channel routing. I hope that bot can be useful to others too! Up until now, many of my projects have moved to Codeberg from SourceHut, but the issue tracker was still stuck on todo.sr.ht due to a lack of a migration tool. I've hacked together srht2forgejo, a tiny script to create Forgejo issues and comments from a todo.sr.ht archive. It's not perfect since the author is the migration user's instead of the original one, but it's good enough. I've now completely migrated all of my projects to Codeberg! I've added a server implementation and tests to go-smee, a small Go library for a Web push forwarding service. It comes in handy when implementing Web push receivers because it's very simple to set up, I've used it when working on nagjo. I've extended the haproxy PROXY protocol to add a new client certificate TLV to relay the raw client certificate from a TLS terminating reverse proxy to a backend server. My goal is enabling client certificate authentication when the soju IRC bouncer sits behind tlstunnel. I've also sent patches for the kimchi HTTP server and go-proxyproto. Because sending a haproxy patch involved git-send-email, I've noticed I've started hitting a long-standing hydroxide signature bug when sending a message. I wasn't previously impacted by this, but some users were. It took a bit of time to hunt down the root cause (some breaking changes in ProtonMail's crypto library), but now it's fixed. Félix Poisot has added two new color management options to Sway: the `color_profile` command now has separate `gamma22` and `srgb` transfer functions (some monitors use one, some use the other), and a `--device-primaries` flag to read color primaries from the EDID (as an alternative to supplying a full ICC profile). With the help of Alexander Orzechowski, we've fixed multiple wlroots issues regarding toplevel capture (aka. window capture) when the toplevel is

completely hidden. It should all work fine now, except one last bug which results in a frozen capture if you're unlucky (aka. you loose the race). I've shipped a number of drmdb improvements. Plane color pipelines are now supported and printed on the snapshot tree and properties table. A warning icon is displayed next to properties which have only been observed on tainted or unstable kernels (as is usually the case for proprietary or vendor kernel modules with custom properties). The device list now shows vendor names for platform devices (extracted from the kernel table). Devices using the new "faux" bus (e.g. vkms) are now properly handled, and all of the possible cursor sizes advertised via the SIZE_HINTS property are now printed. I've also done some SQLite experiments, however they turned out unsuccessful (see that thread and the merge request for more details). delthas has added a new allow_proxy_ip directive to the kimchi HTTP server to mark IP addresses as trusted proxies, and has made it so Forwarded/X-Forwarded-For header fields are not overwritten when the previous hop is a trusted proxy. That way, kimchi can be used in more scenario: behind another HTTP reverse proxy, and behind a TCP proxy which doesn't have a loopback IP address (e.g. tlstunnel in Docker). See you next month!

- [Christian Schaller: Can AI help 'fix' the patent system?](#) (2026/01/21 18:35)

So one thing I think anyone involved with software development for the last decades can see is the problem of "forest of bogus patents". I have recently been trying to use AI to look at patents in various ways. So one idea I had was "could AI help improve the quality of patents and free us from obvious ones?" Lets start with the justification for patents existing at all. The most common argument for the patent system I hear is this one : "Patents require public disclosure of inventions in exchange for protection. Without patents, inventors would keep innovations as trade secrets, slowing overall technological progress.". This reasoning is something that makes sense to me, but it is also screamingly obvious to me that for it to hold true you need to ensure the patents granted are genuinely inventions that otherwise would stay hidden as trade secrets. If you allow patents on things that are obvious to someone skilled in the art, you are not enhancing technological progress, you are hampering it because the next person along will be blocking from doing it. So based on this justification the question then becomes does for example the US Patents Office do a good job filtering out obvious patents? And I believe the answer is "No they don't". Having been working in the space of software for many decades now it is very clear to me that the patent office do very little to avoid patents getting approved for obvious things. And there are many reasons for why that happens I think. First of all if you are an expert in a field you would likely not be working as a case worker in the patent office, no disrespect for the people working there, but for instance if you are a really good software engineer there are much better paying options to be found elsewhere, same goes for medical researchers for example. So we have people who have limited understanding of the field of the patent trying to evaluate if the patent is obvious or not. Secondly the patent office is not really incentivized to deny patents as they are a self funded entity and their 'success' and revenue is based on how many patent applications are submitted. So the lower the bar of approval, the more patents applications get submitted and thus the patent office makes more money and is more "successful". This is a problem, but maybe if we fix the first issue I mention it is maybe possible to reduce the impact of this second issue. So what can AI do to help here? It can help patent reviewers at the patent office with evaluating patents for obviousness due to having been trained on these fields. If we required every patent applied for to be run through a AI engine for review then the patent office reviewer would have to justify why he or she believes something is non-obvious after the AI has looked at it and concluded it is not. My assumption would be that they would very rarely do so. To test out this I decided to ask Gemini to review a H264 video patent that is still valid and which has been used in litigation before, to see if it could do a decent job of reviewing the patent application and judge if this is a true invention or not. Here is what Gemini came up with. US Patent 7,916,782 (Samsung Electronics) is one of the most famous "battleground" patents in the history of smartphone litigation. It has been asserted

against Apple, Huawei, and others. It covers a specific part of the H.264 standard known as CAVLC (Context-Adaptive Variable Length Coding). The Patent: US 7,916,782 : Method and apparatus for video coding and decoding The Mechanism: Efficiently coding the count of zero-valued pixels in a video block ("TotalZeros"). The Problem In video compression, you divide the image into small 4x4 pixel blocks. After processing (transform and quantization), most of the 16 pixels in the block end up being zero. You only have a few non-zero numbers left. Goal: You need to tell the decoder how many zeros are in the block so it can reconstruct the empty space. Constraint: You already told the decoder how many non-zero coefficients exist (let's say you have 5 non-zeros). The "Invention" The patent claims a method where the encoder selects a specific lookup table (VLC table) to encode the "TotalZeros" value, and—crucially—this selection is based on the number of non-zero coefficients (TotalCoeff) that were just processed. If TotalCoeff is 1, use Table A (allows for many zeros). If TotalCoeff is 10, use Table B (allows for fewer zeros). The "Skilled Engineer" Test Imagine you are a video coding engineer in 2002. You are tasked with compressing the "TotalZeros" value. Here is the logical deduction chain you would inevitably follow: The Constraint Check (Basic Math) Engineer's Thought: "I am working with a 4x4 block. That's 16 pixels total." Engineer's Thought: "I just transmitted the number of non-zero pixels. Let's say there are 5 non-zeros. Deduction: "Therefore, the maximum number of zeros possible is $16-5=11$." The Optimization (Common Sense) Engineer's Thought: "I need a Huffman table (lookup table) to compress the TotalZeros value." Engineer's Thought: "Should I use a generic table that goes from 0 to 16?" Deduction: "No, that's wasteful. I know for a fact the value cannot be 12, 13, 14, 15, or 16. If I use a table that includes those impossible numbers, I'm wasting bits on shorter codes for values that will never happen." The Solution (The "Invention") Engineer's Thought: "I should use a smaller, custom table that only contains values 0 through 11." Engineer's Thought: "Actually, I should just make a different table for every possible number of non-zeros. If I have 15 non-zeros, the only possible zero count is 1. I don't even need a table for that; it's implicit!" The Verdict: Valid or Broken? Engineer's View (Obvious): This is not an invention; it is constraint propagation. If you have a container of capacity 16, and you fill it with 5 rocks, it is "obvious" that the remaining space for air is 11. Creating a table sized to "11" isn't a creative spark; it is the mathematical consequence of the block size. To an engineer, failing to do this would be incompetence, not a design choice. Patent Office View (Novel): The USPTO granted this because the patent didn't just claim "optimizing tables." It claimed the specific structure of the solution: Check TotalCoeff. If TotalCoeff is 1, select Table_1. If TotalCoeff is 2, select Table_2. ...and so on. Because no prior paper had explicitly written down "Use these specific tables in this specific order for H.264," it was deemed novel. Conclusion US 7,916,782 is a prime example of the "broken" system because it effectively patents arithmetic efficiency. By granting a monopoly on the logic "Use a small table when you have small space," the patent office allowed Samsung to assert ownership over the only logical way to implement the H.264 standard efficiently. An engineer trying to solve this problem without seeing the patent would essentially be forced to "re-invent" it, because the math leaves no other optimal path. Conclusion: I have a hard time believing a USPTO reviewer would have green lighted this patent after getting this feedback from the AI engine and thus hopefully over time having something like this in place could help us reduce the patent pool to things that genuinely deserve patent protection.

- [Sebastian Wick: Best Practices for Ownership in GLib](#) (2026/01/21 15:31)

For all the rightful criticisms that C gets, GLib does manage to alleviate at least some of it. If we can't use a better language, we should at least make use of all the tools we have in C with GLib. This post looks at the topic of ownership, and also how it applies to libdex fibers. Ownership In normal C usage, it is often not obvious at all if an object that gets returned from a function (either as a real return value or as an out-parameter) is owned by the caller or the callee: `MyThing *thing = my_thing_new ();` If thing is owned by the caller, then the caller also has to release the

object thing. If it is owned by the callee, then the lifetime of the object thing has to be checked against its usage. At this point, the documentation is usually being consulted with the hope that the developer of `my_thing_new` documented it somehow. With gobject-introspection, this documentation is standardized and you can usually read one of these: The caller of the function takes ownership of the data, and is responsible for freeing it. The returned data is owned by the instance. If thing is owned by the caller, the caller now has to release the object or transfer ownership to another place. In normal C usage, both of those are hard issues. For releasing the object, one of two techniques are usually employed: single exit

```
MyThing *thing = my_thing_new ();
gboolean c;
c = my_thing_a (thing);
if (c) c = my_thing_b (thing);
if (c) my_thing_c (thing);
my_thing_release (thing);
/* release thing */
goto cleanup
MyThing *thing = my_thing_new ();
if (!my_thing_a (thing)) goto out;
if (!my_thing_b (thing)) goto out;
my_thing_c (thing);
out: my_thing_release (thing);
/* release thing */
```

Ownership Transfer GLib provides automatic cleanup helpers (`g_auto`, `g_autoptr`, `g_autofd`, `g_autolist`). A macro associates the function to release the object with the type of the object (e.g. `G_DEFINE_AUTOPTR_CLEANUP_FUNC`). If they are being used, the single exit and goto cleanup approaches become unnecessary:

```
g_autoptr(MyThing) thing = my_thing_new ();
if (!my_thing_a (thing)) return;
if (!my_thing_b (thing)) return;
my_thing_c (thing);
```

The nice side effect of using automatic cleanup is that for a reader of the code, the `g_auto` helpers become a definite mark that the variable they are applied on own the object! If we have a function which takes ownership over an object passed in (i.e. the called function will eventually release the resource itself) then in normal C usage this is indistinguishable from a function call which does not take ownership:

```
MyThing *thing = my_thing_new ();
my_thing_finish_thing (thing);
```

If `my_thing_finish_thing` takes ownership, then the code is correct, otherwise it leaks the object thing. On the other hand, if automatic cleanup is used, there is only one correct way to handle either case. A function call which does not take ownership is just a normal function call and the variable `thing` is not modified, so it keeps ownership:

```
g_autoptr(MyThing) thing = my_thing_new ();
my_thing_finish_thing (thing);
```

A function call which takes ownership on the other hand has to unset the variable `thing` to remove ownership from the variable and ensure the cleanup function is not called. This is done by “stealing” the object from the variable:

```
g_autoptr(MyThing) thing = my_thing_new ();
my_thing_finish_thing (g_steal_pointer (&thing));
```

By using `g_steal_pointer` and friends, the ownership transfer becomes obvious in the code, just like ownership of an object by a variable becomes obvious with `g_autoptr`. Ownership Annotations Now you could argue that the `g_autoptr` and `g_steal_pointer` combination without any conditional early exit is functionally exactly the same as the example with the normal C usage, and you would be right. We also need more code and it adds a tiny bit of runtime overhead. I would still argue that it helps readers of the code immensely which makes it an acceptable trade-off in almost all situations. As long as you haven’t profiled and determined the overhead to be problematic, you should always use `g_auto` and `g_steal`! The way I like to look at `g_auto` and `g_steal` is that it is not only a mechanism to release objects and unset variables, but also annotations about the ownership and ownership transfers. Scoping One pattern that is still somewhat pronounced in older code using GLib, is the declaration of all variables at the top of a function:

```
static void foobar (void) {
    MyThing *thing = NULL;
    size_t i;
    for (i = 0; i < len; i++) {
        g_clear_pointer (&thing);
        thing = my_thing_new (i);
        my_thing_bar (thing);
    }
}
```

We can still avoid mixing declarations and code, but we don’t have to do it at the granularity of a function, but of natural scopes:

```
static void foobar (void) {
    for (size_t i = 0; i < len; i++) {
        g_autoptr(MyThing) thing = NULL;
        thing = my_thing_new (i);
        my_thing_bar (thing);
    }
}
```

Similarly, we can introduce our own scopes which can be used to limit how long variables, and thus objects are alive:

```
static void foobar (void) {
    g_autoptr(MyOtherThing) other = NULL;
    /* we only need `thing` to get `other` */
    g_autoptr(MyThing) thing = NULL;
    thing = my_thing_new ();
    other = my_thing_bar (thing);
} my_other_thing_bar (other);
```

Fibers When somewhat complex asynchronous patterns are required in a piece of GLib software, it becomes extremely advantageous to use `libdex` and the system of fibers it provides. They allow writing what looks like

synchronous code, which suspends on await points: `g_autoptr(MyThing) thing = NULL; thing = dex_await_object (my_thing_new_future (), NULL);` If this piece of code doesn't make much sense to you, I suggest reading the libdex Additional Documentation. Unfortunately the await points can also be a bit of a pitfall: the call to `dex_await` is semantically like calling `g_main_loop_run` on the thread default main context. If you use an object which is not owned across an await point, the lifetime of that object becomes critical. Often the lifetime is bound to another object which you might not control in that particular function. In that case, the pointer can point to an already released object when `dex_await` returns: `static DexFuture * foobar (gpointer user_data) { /* foo is owned by the context, so we do not use an autoptr */ MyFoo *foo = context_get_foo (); g_autoptr(MyOtherThing) other = NULL; g_autoptr(MyThing) thing = NULL; thing = my_thing_new (); /* side effect of running g_main_loop_run */ other = dex_await_object (my_thing_bar (thing, foo), NULL); if (!other) return dex_future_new_false (); /* foo here is not owned, and depending on the lifetime * (context might recreate foo in some circumstances), * foo might point to an already released object */ dex_await (my_other_thing_foo_bar (other, foo), NULL); return dex_future_new_true (); }` If we assume that `context_get_foo` returns a different object when the main loop runs, the code above will not work. The fix is simple: own the objects that are being used across await points, or re-acquire an object. The correct choice depends on what semantic is required. We can also combine this with improved scoping to only keep the objects alive for as long as required. Unnecessarily keeping objects alive across await points can keep resource usage high and might have unintended consequences. `static DexFuture * foobar (gpointer user_data) { /* we now own foo */ g_autoptr(MyFoo) foo = g_object_ref (context_get_foo ()); g_autoptr(MyOtherThing) other = NULL; { g_autoptr(MyThing) thing = NULL; thing = my_thing_new (); /* side effect of running g_main_loop_run */ other = dex_await_object (my_thing_bar (thing, foo), NULL); if (!other) return dex_future_new_false (); } /* we own foo, so this always points to a valid object */ dex_await (my_other_thing_bar (other, foo), NULL); return dex_future_new_true (); }` `static DexFuture * foobar (gpointer user_data) { /* we now own foo */ g_autoptr(MyOtherThing) other = NULL; { /* We do not own foo, but we only use it before an * await point. * The scope ensures it is not being used afterwards. */ MyFoo *foo = context_get_foo (); g_autoptr(MyThing) thing = NULL; thing = my_thing_new (); /* side effect of running g_main_loop_run */ other = dex_await_object (my_thing_bar (thing, foo), NULL); if (!other) return dex_future_new_false (); } { MyFoo *foo = context_get_foo (); dex_await (my_other_thing_bar (other, foo), NULL); } return dex_future_new_true (); }` One of the scenarios where re-acquiring an object is necessary, are worker fibers which operate continuously, until the object gets disposed. Now, if this fiber owns the object (i.e. holds a reference to the object), it will never get disposed because the fiber would only finish when the reference it holds gets released, which doesn't happen because it holds a reference. The naive code also suspiciously doesn't have any exit condition. `static DexFuture * foobar (gpointer user_data) { g_autoptr(MyThing) self = g_object_ref (MY_THING (user_data)); for (;;) { g_autoptr(GBytes) bytes = NULL; bytes = dex_await_boxed (my_other_thing_bar (other, foo), NULL); my_thing_write_bytes (self, bytes); } }` So instead of owning the object, we need a way to re-acquire it. A weak-ref is perfect for this. `static DexFuture * foobar (gpointer user_data) { /* g_weak_ref_init in the caller somewhere */ GWeakRef *self_wr = user_data; for (;;) { g_autoptr(GBytes) bytes = NULL; bytes = dex_await_boxed (my_other_thing_bar (other, foo), NULL); { g_autoptr(MyThing) self = g_weak_ref_get (&self_wr); if (!self) return dex_future_new_true (); my_thing_write_bytes (self, bytes); } } }` Conclusion Always use `g_auto/g_steal` helpers to mark ownership and ownership transfers (exceptions do apply) Use scopes to limit the lifetime of objects In fibers, always own objects you need across await points, or re-acquire them

- [Mike Blumenkrantz: 2026 Status](#) (2026/01/14 00:00)

Not A Real Post Still digging myself out of a backlog (and remembering how to computer), so probably no real post this week. I do have some exciting news for the blog though. Now that various public announcements have been made, I can finally reveal the reason why I've been less

active in Mesa of late is because I've been hard at work on Steam Frame. There's a lot of very cool tech involved, and I'm planning to do some rundowns on the software-related projects I've been tackling. Temper your expectations: I won't be discussing anything hardware-related, and there will likely be no mentions of any specific game performance/issues.

- [Sebastian Wick: Improving the Flatpak Graphics Drivers Situation](#) (2026/01/05 23:30)

Graphics drivers in Flatpak have been a bit of a pain point. The drivers have to be built against the runtime to work in the runtime. This usually isn't much of an issue but it breaks down in two cases: If the driver depends on a specific kernel version If the runtime is end-of-life (EOL) The first issue is what the proprietary Nvidia drivers exhibit. A specific user space driver requires a specific kernel driver. For drivers in Mesa, this isn't an issue. In the medium term, we might get lucky here and the Mesa-provided Nova driver might become competitive with the proprietary driver. Not all hardware will be supported though, and some people might need CUDA or other proprietary features, so this problem likely won't go away completely. Currently we have runtime extensions for every Nvidia driver version which gets matched up with the kernel version, but this isn't great. The second issue is even worse, because we don't even have a somewhat working solution to it. A runtime which is EOL doesn't receive updates, and neither does the runtime extension providing GL and Vulkan drivers. New GPU hardware just won't be supported and the software rendering fallback will kick in. How we deal with this is rather primitive: keep updating apps, don't depend on EOL runtimes. This is in general a good strategy. A EOL runtime also doesn't receive security updates, so users should not use them. Users will be users though and if they have a goal which involves running an app which uses an EOL runtime, that's what they will do. From a software archival perspective, it is also desirable to keep things working, even if they should be strongly discouraged. In all those cases, the user most likely still has a working graphics driver, just not in the flatpak runtime, but on the host system. So one naturally asks oneself: why not just use that driver? That's a load-bearing "just". Let's explore our options. Exploration Attempt #1: Bind mount the drivers into the runtime. Cool, we got the driver's shared libraries and ICDs from the host in the runtime. If we run a program, it might work. It might also not work. The shared libraries have dependencies and because we are in a completely different runtime than the host, they most likely will be mismatched. Yikes. Attempt #2: Bind mount the dependencies. We got all the dependencies of the driver in the runtime. They are satisfied and the driver will work. But your app most likely won't. It has dependencies that we just changed under its nose. Yikes. Attempt #3: Linker magic. Until here everything is pretty obvious, but it turns out that linkers are actually quite capable and support what's called linker namespaces. In a single process one can load two completely different sets of shared libraries which will not interfere with each other. We can bind mount the host shared libraries into the runtime, and dlopen the driver into its own namespace. This is exactly what libcapsule does. It does have some issues though, one being that the libc can't be loaded into multiple linker namespaces because it manages global resources. We can use the runtime's libc, but the host driver might require a newer libc. We can use the host libc, but now we contaminate the apps linker namespace with a dependency from the host. Attempt #4: Virtualization. All of the previous attempts try to load the host shared objects into the app. Besides the issues mentioned above, this has a few more fundamental issues: The Flatpak runtimes support i386 apps; those would require a i386 driver on the host, but modern systems only ship amd64 code. We might want to support emulation of other architectures later It leaks an awful lot of the host system into the sandbox It breaks the strict separation of the host system and the runtime If we avoid getting code from the host into the runtime, all of those issues just go away, and GPU virtualization via Virtio-GPU with Venus allows us to do exactly that. The VM uses the Venus driver to record and serialize the Vulkan commands, sends them to the hypervisor via the virtio-gpu kernel driver. The host uses virglrenderer to deserializes and executes the commands. This makes sense for VMs, but we don't have a VM, and we might not have the virtio-gpu kernel module, and we might not be able to load it without

privileges. Not great. It turns out however that the developers of virglrenderer also don't want to have to run a VM to run and test their project and thus added vtest, which uses a unix socket to transport the commands from the mesa Venus driver to virglrenderer. It also turns out that I'm not the first one who noticed this, and there is some glue code which allows Podman to make use of virgl. You can most likely test this approach right now on your system by running two commands: `rendernodes=(/dev/dri/render*) virgl_test_server --venus --use-gles --socket-path /tmp/flatpak-virgl.sock --rendernode "${rendernodes[0]}" & flatpak run --nodevice=dri --filesystem=/tmp/flatpak-virgl.sock --env=VN_DEBUG=vtest --env=VTEST_SOCKET_NAME=/tmp/flatpak-virgl.sock org.gnome.clocks` If we integrate this well, the existing driver selection will ensure that this virtualization path is only used if there isn't a suitable driver in the runtime. Implementation Obviously the commands above are a hack. Flatpak should automatically do all of this, based on the availability of the dri permission. We actually already start a host program and stop it when the app exits: `xdg-dbus-proxy`. It's a bit involved because we have to wait for the program (in our case `virgl_test_server`) to provide the service before starting the app. We also have to shut it down when the app exits, but flatpak is not a supervisor. You won't see it in the output of `ps` because it just execs bubblewrap (`bwrap`) and ceases to exist before the app even started. So instead we have to use the kernel's automatic cleanup of kernel resources to signal to `virgl_test_server` that it is time to shut down. The way this is usually done is via a so called sync fd. If you have a pipe and poll the file descriptor of one end, it becomes readable as soon as the other end writes to it, or the file description is closed. Bubblewrap supports this kind of sync fd: you can hand in a one end of a pipe and it ensures the kernel will close the fd once the app exits. One small problem: only one of those sync fds is supported in `bwrap` at the moment, but we can add support for multiple in Bubblewrap and Flatpak. For waiting for the service to start, we can reuse the same pipe, but write to the other end in the service, and wait for the fd to become readable in Flatpak, before exec'ing `bwrap` with the same fd. Also not too much code. Finally, virglrenderer needs to learn how to use a sync fd. Also pretty trivial. There is an older MR which adds something similar for the Podman hook, but it misses the code which allows Flatpak to wait for the service to come up, and it never got merged. Overall, this is pretty straight forward. Conclusion The virtualization approach should be a robust fallback for all the cases where we don't get a working GPU driver in the Flatpak runtime, but there are a bunch of issues and unknowns as well. It is not entirely clear how forwards and backwards compatible vtest is, if it even is supposed to be used in production, and if it provides a strong security boundary. None of that is a fundamental issue though and we could work out those issues. It's also not optimal to start `virgl_test_server` for every Flatpak app instance. Given that we're trying to move away from blanket dri access to a more granular and dynamic access to GPU hardware via a new daemon, it might make sense to use this new daemon to start the `virgl_test_server` on demand and only for allowed devices.

- [Timur Kristóf: A love song for Linux gamers with old GPUs \(EOY 2025\)](#) (2026/01/01 00:00)

AMD GPUs are famous for working very well on Linux. However, what about the very first GCN GPUs? Are they working as well as the new ones? In this post, I'm going to summarize how well these old GPUs are supported and what I've been doing to improve them. This story is about the first two generations of GCN: Southern Islands (aka. SI, GCN1, GFX6) and Sea Islands (aka. CIK, GCN2, GFX7). Working on old GPUs While AMD GPUs generally have a good reputation on Linux, these old GCN graphics cards have been a sore spot for as long as I've been working on the driver stack. It occurred to me that resolving some of the long-standing issues on these old GPUs might be a great way to get me started on working on the `amdgpu` kernel driver and would help improve the default user experience of Linux users on these GPUs. I figured that it would give me a good base understanding, and later I could also start contributing code and bug fixes to newer GPUs. Where I started The RADV team has supported RADV on SI and CIK GPUs for a long time. RADV support was already there even before I joined the team in mid-2019. Daniel

added ACO support for GFX7 in November 2019, and Samuel added ACO support for GFX6 in January 2020. More recently, Martin added a Tahiti (GFX6) and Hawaii (GFX7) GPU to the Mesa CI which are running post-merge “nightly” jobs. So we can catch regressions and test our work on these GPUs quite quickly. The kernel driver situation was less fortunate. On the kernel side, amdgpu (the newer kernel driver) has supported CIK since June 2015 and SI since August 2016. DC (the new display driver) has supported CIK since September 2017 (the beginning), and SI support was added in July 2020 by Mauro. However, the old radeon driver was the default driver. Unfortunately, radeon doesn’t support Vulkan, so in the default user experience, users couldn’t play most games or benefit from any of the Linux gaming related work we’ve been doing for the last 10 years. In order to get working Vulkan support on SI and CIK, we needed to use the following kernel params: `radeon.si_support=0 radeon.cik_support=0 amdgpu.si_support=1 amdgpu.cik_support=1` Then, you could boot with amdgpu and enjoy a semblance of a good user experience until the GPU crashed / hung, or until you tried to use some functionality which was missing from amdgpu, or until you plugged in a display which the display driver couldn’t handle. It was... not the best user experience. Where to go from there? The first question that came to mind is, why wasn’t amdgpu the default kernel driver for these GPUs? Since the “experimental” support had been there for 10 years, we had thought the kernel devs would eventually just enable amdgpu by default, but that never happened. At XDC 2024 I met Alex Deucher, the lead developer of amdgpu and asked him what was missing. Alex explained to me that the main reason the default wasn’t switched is to avoid regressions for users who rely on some functionality not supported by amdgpu: Display features: analog connectors in DC (or DP/HDMI audio support in non-DC) VCE1 for video coding on SI It doesn’t seem like much, does it? How hard can it be? Display features On a 2025 summer afternoon... I messaged Alex Deucher to get some advice on where to start. Alex was very considerate and helped me to get a good understanding of how the code is organized, how the parts fit together and where I should start reading. Harry Wentland also helped a lot with making a plan how to fit analog connectors in DC. Then, I plugged my monitors into my Raphael iGPU to be used as a primary GPU, then plugged in an old Oland card as a secondary GPU, and started hacking. Focus For the display, I decided that the best way forward is to add what is missing from DC for these GPUs and use DC by default. That way, we can eventually get rid of the legacy display code (which was always meant as a temporary solution until DC landed). Additionally, I decided to focus on dedicated GPUs because these are the most useful for gaming and are easy to test using a desktop computer. There is still work left to do for CIK APUs. Analog connector support in DC Analog connectors have been actually quite easy to deal with, once I understood the structure of the DC (display core) codebase. I could use the legacy display code as a reference. The DAC (digital to analog converter) is actually programmed by the VBIOS, and the driver just needs to call the VBIOS to tell it what to do. Easier said than done, but not too hard. It also turned out that some chips that already defaulted to DC (eg. Tonga, Hawaii) also have analog connectors, which apparently just didn’t work on Linux by default. I managed to submit the first version of this in July. Then I was sidetracked with a lot of other issues, so I submitted the second version of the series in September, which then got merged. Going shopping It is incredibly difficult to debug issues when you don’t have the hardware to reproduce them yourself. Some developers have a good talent for writing patches to fix issues without actually seeing the issue, but I feel I still have a long way to go to be that good. It was pretty clear from the beginning that the only way to make sure my work actually works on all SI/CIK GPUs is to test all of them myself. So, I went ahead and acquired at least one of each SI and CIK chip. I got most of them from used hardware ad sites, and Leonardo Frassetto sent me a few as well. Fixing DC support on SI (DCE6) After I got the analog connector working using the old GPUs as a secondary GPU, I thought it’s time to test how well it works as a primary GPU. You know, the way most actual users would use them. So I disabled the iGPU and booted my computer with each dGPU with `amdgpu.dc=1` to see what happens. This is where things started going crazy... Tahiti (R9 280X) booted into “no signal”, absolutely dead Oland

(R7 250) had heavy flickering and hung very quickly Oland (Radeon 520) booted into “unsupported signal” with DC and massive flickering with non-DC Pitcairn (R9 270X) had some artifacts Cape Verde (HD 7770) I didn’t even plug it in at this point... Hainan fortunately doesn’t have DCE (display controller engine) so that wasn’t a problem The way to debug these problems is the following: Boot with `amdgpu.dc=0`, dump all DCE registers using `umr: umr -r oland.dce600.* > oland_nodc_good.txt` Boot with `amdgpu.dc=1`, dump all DCE registers using `umr: umr -r oland.dce600.* > oland_dc_bad.txt` Compare the two DCE register dumps using a diff viewer, eg. Meld. Try to find what are the register differences that are responsible for the bad behaviour. Use `umr` (either the GUI or CLI) to try to change the registers in real time, poke at it until the issue is gone. Wait until headache is gone. Read the code that sets the problematic registers and develop an actual fix. I decided to fix the bugs before adding new features. I sent a few patch series to address a bunch of display issues mainly with SI (DCE6): Fixed broken PLL programming and some mistakes Fixed DC “overclocking” the display clock and a few other issues — many years ago someone fixed an issue on Polaris by unconditionally raising the display clock by 15%, but unfortunately they also applied this to older GPUs; additionally the display clock was already set higher than the max Fixed DVI-D/HDMI adapters — these would just give a black screen when I plugged in a 4K monitor While at it, I also fixed them in the legacy code Fixed a freeze caused by relying on a non-existent interrupt — it seems that DCE6 is not capable of VRR, so we just shouldn’t try to enable it or rely on interrupts that don’t exist on this HW Fixed another black screen by rejecting too high pixel clocks — technically, DP supports the bandwidth required by 4K 120Hz using 6-bit color with YUV420 on SI/CIK/VI, so DC would happily advertise this mode, but the GPUs didn’t actually support a high enough display clock for 4K 120Hz Fixed an issue with the DCE6 scaler not being properly disabled — apparently the VBIOS sets up the scaler automatically on some GPUs, which needs to be disabled by the kernel, otherwise you get weird artifacts DisplayPort/HDMI audio support on SI (DCE6) I noticed that HDMI audio worked alright on all GPUs with DC (as expected), however DP audio didn’t (which was unexpected). However it worked when both DP and HDMI were plugged in... After consulting with Alex and doing some trial and error, it turned out that this was just due to setting some clock frequency in the wrong way: Fix DP audio DTO1 clock source on DCE6. In order to figure out the correct frequencies, I wrote a script that set the frequency using `umr` then played a sound. I just laid back and let the script run until I heard the sound. Then it was just a matter of figuring out why that frequency was correct. A small fun fact: it turns out that DP audio on Tahiti didn’t work on any Linux driver before. Now it works with DC. Poweeeeer The DCE (Display Controller Engine), just like other parts of the GPU, has its own power requirements and needs certain clocks, voltages, etc. It is the responsibility of the power management code to make sure DCE gets the power it needs. Unfortunately, DC didn’t talk to the legacy power management code. Even more unfortunately, the power management code was buggy so that’s what I started with. SI power management fixes — contains some fixes to get Tahiti to boot with DC, also it turns out that the SMC (system management controller) needs a longer timeout to complete some tasks More SI power management and PLL fixes — most importantly this disables ASPM on SI, which caused “random hangs” Finally, this series hooks up SI to the legacy DPM code — this is required because the power management code needs to know how many and what kind of displays are connected After I was mostly done with SI, I also fixed an issue with CIK, where the shutdown temperature was incorrectly reported. VCE1 video encoding on SI Video encoding is usually an afterthought, not something that most users think about unless they are interested in streaming or video transcoding. It was definitely an afterthought for the hardware designers of SI, which has the first generation VCE (video coding engine) that only supports H264 and only up to 2048 x 1152. However, the old radeon kernel driver supports this engine and for anyone relying on this functionality, it would be a regression when switching to `amdgpu`. So we need to support it. There was already some work by Alexandre Demers to support VCE1, but that work was stalled due to issues caused by the firmware validation mechanism. In order to switch SI to `amdgpu` by default, I needed to deal with

VCE1. So I started a conversation with Christian König (amdgpu expert) to identify what the problem actually was, and with Alexandre to see how far along his work was. It turns out that due to some HW/FW limitations, the firmware (VCPU BO) needs to be mapped at a low 32-bit address. It also needs to be in VRAM for optimal performance (otherwise it would incur too many roundtrips to system RAM). However, there was no way for amdgpu to place something in VRAM and map it in the low 32-bit address space. (Note, it can actually do this for the address space of userspace apps, just not inside the kernel itself.) Christian helped me a lot with understanding how the memory controller and the page table work. After I got over the headache, I came up with this idea: Let amdgpu place the VCPU BO in VRAM Map the GART (graphics address remapping table) in the low 32-bit address space (instead of using best fit) Insert a few page table entries in the GART which would practically map the VCPU BO into the low 32-bit address space With that out of the way, the rest of the work on VCE1 was pretty straightforward. I could use Alexandre's research, as well as the VCE2 code from amdgpu and the VCE1 code from radeon as a reference. Finally, a few reviews and three revisions later, the VCE1 series was accepted. Final thoughts Who is this for? In the current economic situation of our world, I expect that people are going to use GPUs for much longer, and replace them less often. And when an old GPU is replaced, it doesn't die, it goes to somebody who upgrades an even older GPU. Eventually it will reach somebody that can't afford a better one. There are some efforts to use Linux to keep old computers alive, for example this one. My goal with this work is to make Linux gaming a good experience also for those who use old GPUs. Other than that, I also did it for myself. Not because I want to run old GPUs myself, but because it has been a great learning experience to get into the amdgpu kernel driver. Why amdgpu? Why DC? The open source community including AMD themselves as well as other entities like Valve, Igalia, Red Hat etc. have invested a lot of time and effort into amdgpu and DC, which now support many generations of AMD GPUs: GCN1-5, RDNA1-4, as well as CDNA. In fact amdgpu supports more generations of GPUs now than what came before, and it looks like it will support many generations of future GPUs. By making amdgpu work well with SI and CIK, we ensure that these GPUs remain competently supported for the foreseeable future. By switching SI and CIK to use DC by default, we enable display features like atomic modesetting, VRR, HDR, etc. and this also allows the amdgpu maintainers to eventually move on from the legacy display code without losing functionality. What is left to do? Now that amdgpu is at feature parity with radeon on old GPUs, we switched the default to amdgpu on SI and CIK dedicated GPUs. It's time to start thinking about what else is left to do. Add support for DRM format modifiers for all SI/CIK/VI/Polaris GPUs. This would be a huge step forward for the Vulkan ecosystem, it would enable using purely Vulkan based compositors, Zink, and other features. Add support for TRAVIS and NUTMEG display bridges, so that we can also switch to amdgpu by default for CIK APUs. I couldn't find the hardware for this work (mainly looking for Kaveri APUs), if you have it and want to help, please reach out. Your dmesg log will mention if the APU uses TRAVIS or NUTMEG. Refactor SI and KV power management so that we can retire the legacy power management code, which would further ease the maintenance burden of these GPUs. Eventually retire the non-DC legacy display code from amdgpu to ease the maintenance burden. Deal with a few lingering bugs, such as power limit on Radeon 430, black screen with the analog connector on Radeon HD 7790, as well as reenable compute queues on SI, mitigate VM faults on SI/CIK, etc. Verify sparse mapping (PRT) support. I already wrote a kernel fix and a Mesa MR for enabling it. Implement transfer queue support for old GPUs in RADV. What have I learned from all this? It isn't that scary Kernel development is not as scary as it looks. It is a different technical challenge than what I was used to, but not in a bad way. Just needed to figure out a good workflow for how to configure a code editor, as well as what is a good way to test my work without rebuilding everything all the time. Maintainers are friendly AMD engineers have been very friendly and helpful to me all the way. Although there are a lot of memes and articles on the internet about attitude and rude/toxic messages by some kernel developers, I didn't see that in amdgpu at least. My approach was that even before I wrote a single line of code, I started talking to the maintainers (who would

eventually review my patches) to find out what would be the good solution to them and how to get my work accepted. Communicating with the maintainers saved a lot of time and made the work faster, more pleasant and more collaborative. Development latency Sadly, there is a huge latency between a Linux kernel developer working on something and the work reaching end users. Even if the patches are accepted quickly, it can take 3~6 months until users can actually use it. In Mesa, we can merge any features to the next Mesa release up to the branch point, and afterwards we backport bug fixes to that release. Simple and efficient. In the Linux kernel, there is a deadline for merging new features to the next release, (it's not clearly communicated when that is). Bug fixes may be backported to previous releases any time, but upstream maintainers aren't involved in that. In hindsight, if I had focused on finishing the analog support and VCE1 first (instead of fixing all the bugs I found), my work would have ended up in Linux 6.18 (including the bug fixes, as there is no deadline for those). Due to how I prioritized bug fixing, the features I've developed are only included in Linux 6.19, so this will be the version where SI and CIK will default to amdgpu by default. XDC 2025 I presented a lightning talk on this topic at XDC 2025, where I talked about the state of SI and CIK support as of September 2025. You can find the slide deck here and the video here. Acknowledgements I'd like to say a big thank you to all of these people. All of the work I mentioned in this post would not have been possible without them. Alex Deucher, Christian König (amdgpu devs) Harry Wentland, Rodrigo Siquiera (DC devs) Marek Olsák, Pierre-Eric Pelloux-Prayer (radeonsi devs) Bas Nieuwenhuizen, Samuel Pitoiset (radv devs) Martin Roukala, né Peres (CI expert) Tom St Dennis (umr dev) Mauro Rossi (DCE6 in DC) Alexandre Demers (VCE1 research) Leonardo Frassetto (HW donation) Roman Elshin and others (testing) Pierre-Loup Griffais (Valve) Which graphics cards are affected exactly? When in doubt, consult Wikipedia. GFX6 aka. GCN1 - Southern Islands (SI) dedicated GPUs: amdgpu is now the default kernel driver as of Linux 6.19. The DC display driver is now the default and is now usable for these GPUs. DC now supports analog connectors, power management is less buggy, and video encoding is now supported by amdgpu. Tahiti Radeon HD 7870 XT, 7950, 7970, 7990, 8950, 8970, 8990 Radeon R9 280, 280X FirePro W8000, W9000, D500, D700, S9000, S9050, S10000 Radeon Sky 700, 900 Pitcairn Radeon HD 7850, 7870, 7970M, 8870, 8970M Radeon R9 265, 270, 270X, 370, 370X, M290X, M390 FirePro W5000, W7000, D300, R5000, S7000 Cape Verde Radeon HD 7730, 7750, 7770, 8730, 8760 Radeon R7 250E, 250X, 350, 450 FirePro W600, W4100, M4000, M6000 Oland Radeon HD 8570, 8670 Radeon R5 240, 250, 330, 340, 350, 430, 520, 610 FirePro W2100 various mobile GPUs Hainan various mobile GPUs GFX7 aka. GCN2 - Sea Islands (CIK) dedicated GPUs: amdgpu is now the default kernel driver as of Linux 6.19. The DC display driver is now the default for Bonaire (was already the case for Hawaii). DC now supports analog connectors. Minor bug fixes. Hawaii Radeon R9 290, 290X, 295X2, 390, 390X FirePro W8100, W9100, S9100, S9150, S9170 Bonaire Radeon HD 7790/8870 Radeon R7 260/360/450 Radeon RX 455, FirePro W5100, etc. various mobile GPUs GFX8 aka. GCN3 - Volcanic Islands (VI) dedicated GPUs: DC now supports analog connectors. (Note that amdgpu and DC were already supported on these GPUs since release.) Tonga Radeon R9 285, 380, 380X (other chips of this family are not affected by the work in this post)

- [Lennart Poettering: Mastodon Stories for systemd v259](#) (2025/12/30 23:00)

On Dec 17 we released systemd v259 into the wild. In the weeks leading up to that release (and since then) I have posted a series of serieses of posts to Mastodon about key new features in this release, under the #systemd259 hash tag. In case you aren't using Mastodon, but would like to read up, here's a list of all 25 posts: Post #1: systemd-resolved Hooks Post #2: dlopen() everything Post #3: systemd-analyze dlopen-metadata Post #4: run0 --empower Post #5: systemd-vmspawn --bind-user= Post #6: Musl libc support Post #7: systemd-repart without device name Post #8: Parallel kmod loading in systemd-modules-load.service Post #9: NvPCR Support Post #10: systemd-analyze nvcpcrs Post #11: systemd-repart Varlink IPC API Post #12: systemd-vmspawn block device serial Post #13: systemd-repart --defer-partitions-empty= + --defer-partitions-

factory-reset= Post #14: userdb support for UUID queries Post #15: Wallclock time in service completion logging Post #16: systemd-firstboot -- prompt-keymap-auto Post #17: \$LISTEN_PIDFDID Post #18: Incremental partition rescanning Post #19: ExecReloadPost= Post #20: Transaction order cycle tracking Post #21: systemd-firstboot facelift Post #22: Per-User systemd-machined + systemd-importd Post #23: systemd-udev's OPTIONS="dump-json" Post #24: systemd-resolved's DumpDNSConfiguration() IPC Call Post #25: DHCP Server EmitDomain= + Domain= I intend to do a similar series of serieses of posts for the next systemd release (v260), hence if you haven't left tech Twitter for Mastodon yet, now is the opportunity. My series for v260 will begin in a few weeks most likely, under the #systemd260 hash tag. In case you are interested, here is the corresponding blog story for systemd v258, here for v257, and here for v256.

- [Timur Kristóf: Understanding your Linux open source drivers](#) (2025/12/21 23:52)

After introducing how graphics drivers work in general, I'd like to give a brief overview about what is what in the Linux graphics stack, what are the important parts and what the key projects are where the development happens, as well as what you need to do to get the best user experience out of it. The open source Linux graphics driver stack Please refer to my previous post for a more detailed general explanation on graphics drivers in general. This post focuses on how things work in the open source graphics stack on Linux. Which GPUs are supported? We have open source drivers for the GPUs from all major manufacturers with varying degrees of success. Some companies (eg. AMD, Intel and others) choose to participate in the open source community and develop open source drivers themselves, with contributions also coming from other parties (eg. Valve, Red Hat, Google, etc). If you have an AMD or Intel GPU, their open source drivers typically work better than their alternatives (if any), and have been for a long time. Others (eg. NVidia and Apple) don't recognize the benefits of this style of development and leave it to the community to create drivers based on reverse-engineering or sometimes minimal help from the manufacturer. If you have an NVidia GPU, as of late 2025 the open source drivers may not be quite ready just yet. In this post I am not going to discuss the proprietary drivers. What parts do you need? The components you need in order to get your GPU working on open source drivers on a Linux distro are the following: Linux kernel (obviously) — contains the open source kernel drivers (KMD). linux-firmware — contains the necessary firmware. Note that even when the actual drivers are open source, most (all?) GPUs require closed source firmware to function. Mesa — contains the most userspace drivers (relevant to gaming) as well as a shader compiler stack. LLVM — needed by some Mesa drivers for shader compilation. Some vendors have other projects (eg. AMD ROCm) for supporting other features that aren't part of Mesa. These parts are out of scope for this blog post. To make your GPU work, you need new enough versions of the Linux kernel, linux-firmware and Mesa (and LLVM) that include support for your GPU. To make your GPU work well, I highly recommend to use the latest stable versions of all of the above. If you use old versions, you are missing out. By using old versions you are choosing not to benefit from the latest developments (features and bug fixes) that open source developers have worked on, and you will have a sub-par experience (especially on relatively new hardware). Wait, aren't the drivers in the kernel? If you read Reddit posts, you will stumble upon some people who believe that "the drivers are in the kernel" on Linux. This is a half-truth. Only the KMDs are part of the kernel, everything else (linux-firmware, Mesa, LLVM) is distributed in separate packages. How exactly those packages are organized, depends on your distribution. What is the Mesa project? Mesa is a collection of userspace drivers which implement various different APIs. It is the cornerstone of the open source graphics stack. I'm going to attempt to give a brief overview of what are the most relevant parts of Mesa. Gallium An important part of Mesa is the Gallium driver infrastructure, which contains a lot of common code for implementing different APIs, such as: Graphics: OpenGL, OpenGL ES, EGL Compute: OpenCL Video decoding and encoding: VA-API (previously also VDPAU) Vulkan Mesa also contains a collection of Vulkan drivers. Originally, Vulkan was deemed "lower level than Gallium", so Vulkan drivers are not part of the

Gallium driver infrastructure. However, Vulkan has a lot of overlapping functionality with the aforementioned APIs, so Vulkan drivers still share a lot of code with their Gallium counterparts when appropriate. NIR Another important part of Mesa is the NIR shader compiler stack, which is at the heart of every Mesa driver that is still being maintained. This enables sharing a lot of compiler code across different drivers. I highly recommend Faith Ekstrand's post In defense of NIR to learn more about it. Compatibility layers and API translation Technically they are not drivers, but in practice, if you want to run Windows games, you will need a compatibility layer like Wine or Proton, including graphics translation layers. The recommended ones are: DXVK: translates DirectX 8-11 to Vulkan. VKD3D-Proton: translates DirectX 12 to Vulkan. Those are default in Proton and offer the best performance. However, for "political" reasons, these are sadly not the defaults in Wine, so either you'll have to use Proton or make sure to install the above in Wine manually. Just for the sake of completeness, I'll also mention the Wine defaults: WineD3D: translates DirectX 1-11 to OpenGL. It is what we all used before Vulkan and DXVK existed, sadly its performance and compatibility has always been lacking. VKD3D: translates DirectX 12 to Vulkan. Not practically usable. Can only actually run a select few games, and those with lackluster performance. (Not to be confused with VKD3D-Proton, which is actually fully-featured.) Side note about window systems Despite the X server being abandoned for a long time, there is still a debate between Linux users whether to use a Wayland compositor or the X server. I'm not going to discuss the advantages and disadvantages of these, because I don't participate in their development and I feel it has already been well-explained by the community. I'm just going to say that it helps to choose a competent compositor that implements direct scanout. This means that the frames as rendered by your game can be sent directly to the display without the need for the compositor to do any additional processing on it. In this blog post I focus on just the driver stack, because that is largely shared between both solutions. Making your games run (well) Sadly, many Linux distributions choose to ship old versions of the kernel and/or other parts of the driver stack, giving their users a sub-par experience. Debian, Ubuntu LTS and their derivatives like Mint, Pop OS, etc. are all guilty of this. They justify this by claiming that older versions are more reliable, but this is actually not true. In reality, us driver developers as well as the developers of the API translation layers work hard to implement new features that are needed to get new games working, as well as fixing bugs that are exposed by new games (or updates of old games). Regressions are real, but they are usually quickly fixed thanks to the extensive testing that we do: every time we merge new code, our automated testing system runs the full Vulkan conformance test suite to make sure that all functionality is still intact, thanks to Martin's genius.

- [Simon Ser: Status update, December 2025](#) (2025/12/21 22:00)

Hi all! This month the new KMS plane color pipeline API has finally been merged! It took multiple years and continued work and review by engineers from multiple organizations, but at last we managed to push it over the finish line. This new API exposes to user-space new hardware blocks: these applying color transformations before blending multiple KMS planes as a final composited image to be sent on the wire. This API unlocks power-efficient and low-latency color management features such as HDR. Still, much remains to be done. Color pipelines are now exposed on AMD and VKMS, Intel and other vendors are still working on their driver implementation. Melissa Wen has written a `drm_info` patch to show pipeline information, some more work is needed to plumb it through `drmdb`. Some patches have been floated to leverage color pipelines for post-blending transforms too (currently KMS only supports a fixed rudimentary post-blending pipeline with two LUTs and one 3x3 matrix). On the wlrroots side, Félix Poisot has redesigned the way post-blending color transforms are applied by the renderer. The API used to be a mix of descriptive (describing which primaries and transfer functions the output buffer uses) and prescriptive (passing a list of operations to apply). Now it's fully prescriptive, which will help for offloading these transformations to the DRM backend. GnSight has contributed support for the `wlr-foreign-toplevel-management-v1` protocol to the Cage kiosk compositor. This enables better control over windows running inside the compositor:

external tools can close or bring windows to the front. mhorky has added client support for one-way method calls to go-varlink, as well as a nice Registry enhancement to add support for the org.varlink.service interface for free, for discovery and introspection of Varlink services. Now that the module is feature-complete I've released version 0.1.0. delthas has introduced support for authenticating with the soju IRC bouncer via TLS client certificates. He has contributed a simple audio recorder to the Goguma mobile IRC client, plus new buttons above the reaction list to be able to easily +1 another user's reaction. Hubert Hirtz has sent a collection of bug fixes and has added a button to reveal the password field contents on the connection screen. I've resurrected work on some old projects I'd almost forgotten about. I've pushed a few patches for libicc, adding support for encoding multi-process transforms, luminance and metadata. I've added a basic test suite to libjsonschema, and improved handling of objects and arrays without enough information to automatically generate types from. But the old project I've spent most of my time on is go-mls, a Go implementation of the Messaging Layer Security (MLS) protocol. MLS is an end-to-end encryption protocol for chat messages. My goal is twofold: learn how MLS works under the hood (implementing something is one of the best ways for me to understand that something), and lay the groundwork for a future end-to-end encryption IRC extension. This month I've fixed up the remaining failures in the test suite and I've implemented just enough to be able to create a group, add members to it, and exchange an encrypted message. I'll work on remaining group operations (e.g. removing a member) next. Last, I've migrated FreeDesktop's Mailman 3 installation to PostgreSQL from SQLite. Mailman 3's SQLite integration had pretty severe performance issues, these are gone with PostgreSQL. The migration wasn't straightforward: there is no tooling to migrate Mailman 3 core's data between database engines, so I had to manually fill the new database with the old data. I've migrated two more mailing lists to Mailman 3: fhs and nouveau. I plan to continue the migration in the coming months, and hopefully we'll be able to decommission Mailman 2 in a not-so-distant future. See you next year!

- [Timur Kristóf: How do graphics drivers work?](#) (2025/12/16 00:09)

I'd like to give an overview on how graphics drivers work in general, and then write a little bit about the Linux graphics stack for AMD GPUs. The intention of this post is to clear up a bunch of misunderstandings that people have on the internet about open source graphics drivers. What is a graphics driver? A graphics driver is a piece of software code that is written for the purpose of allowing programs on your computer to access the features of your GPU. Every GPU is different and may have different capabilities or different ways of achieving things, so they need different drivers, or at least different code paths in a driver that may handle multiple GPUs from the same vendor and/or the same hardware generation. The main motivation for graphics drivers is to allow applications to utilize your hardware efficiently. This enables games to render pretty pixels, scientific apps to calculate stuff, as well as video apps to encode / decode efficiently. Organization of graphics drivers Compared to drivers for other hardware, graphics is very complicated because the functionality is very broad and the differences between each piece of hardware can be also vast. Here is a simplified explanation on how a graphics driver stack usually works. Note that most of the time, these components (or some variation) are bundled together to make them easier to use. GPU firmware (FW) — low-level code for power management, context switching, command processing, display engine, video encoding/decoding, etc. Kernel driver, aka. kernel-mode driver (KMD) — makes it possible for multiple userspace applications to submit commands to the GPU, and is responsible for memory management and display functionality. Userspace driver, aka. user-mode driver (UMD) — responsible for implementing an API, such as Vulkan, OpenGL, etc. For each piece of hardware, there may be multiple different UMDs implementing different APIs. Shader compiler — a userspace library that compiles shader programs for your GPU from the HW-independent code that applications have. Can be possibly shared between UMDs, sometimes developed as a separate project. I'll give a brief overview of each component below. GPU firmware Most GPUs have additional processors (other than the shader cores)

which run a firmware that is responsible for operating the low-level details of the hardware, usually stuff that is too low-level even for the kernel. The firmware on those processors are responsible for: power management, context switching, command processing, display, video encoding/decoding etc. Among other things it parses the commands we submitted to it, launches shaders, distributes work between the shader cores etc. Some GPU manufacturers are moving more and more functionality to firmware, which means that the GPU can operate more autonomously and less intervention is needed by the CPU. This tendency is generally positive for reducing CPU time spent on programming the GPU (as well as “CPU bubbles”), but at the same time it also means that the way the GPU actually works becomes less transparent. Kernel driver You might ask, why not implement all driver functionality in the kernel? Wouldn't it be simpler to “just” have everything in the kernel? The answer is no, mainly because there is a LOT going on which nobody wants in the kernel. You don't want to have your kernel crash when a game misbehaves. Sadly it can still happen, but it would happen a lot more if the kernel and userspace components weren't separated. You definitely don't want to run a fully-fledged compiler inside your kernel which takes arbitrary input from the user. You want to avoid having to upgrade your kernel to deploy most fixes and improvements to the graphics stack. (This is not always avoidable but can be minimized.) So, usually, the KMD is only left with some low-level tasks that every user needs: Command submission userspace API: an interface that allows userspace processes to submit commands to the GPU, query information about the GPU, etc. Memory management: deciding which process gets to use how much VRAM, defining GTT, handling low-memory situations, etc. Display functionality: making display connectors work, by programming the registers of the display controller. There is also a separate uAPI for just this purpose. Power management: making sure the GPU doesn't draw too much power when not needed, and also making sure applications can get the best clock speeds etc. when that is needed, in cooperation with the power management firmware. GPU recovery: when the GPU hangs or crashes for some reason, it's the kernel's responsibility to ensure that the GPU can be recovered and that the crash doesn't affect other processes. Userspace driver Applications interact with userspace drivers instead of the kernel (or the hardware directly). Userspace drivers are compiled as shared libraries and are responsible for implementing one or more specific APIs for graphics, compute or video for a specific family of GPUs. (For example, Vulkan, OpenGL or OpenCL, etc.) Each graphics API has entry points which load the available driver(s) for the GPU(s) in the user's system. The Vulkan loader is an example of this; other APIs have similar components for this purpose. The main functionality of a userspace driver is to take the commands from the API (for example, draw calls or compute dispatches) and turn them into low level commands in a binary format that the GPU can understand. In Vulkan, this is analogous to recording a command buffer. Additionally, they utilize a shader compiler to turn a higher level shader language (eg. GLSL) or bytecode (eg. SPIR-V) into hardware instructions which the GPU's shader cores can execute. Furthermore, userspace drivers also take part in memory management, they basically act as an interface between the memory model of the graphics API and kernel's memory manager. The userspace driver calls the aforementioned kernel uAPI to submit the recorded commands to the kernel which then schedules it and hands it to the firmware to be executed. Shader compiler If you've seen a loading screen in your favourite game which told you it was “compiling shaders...” you probably wondered what that's about and why it's necessary. Unlike CPUs which have converged to a few common instruction set architectures (ISA), GPUs are a mess and don't share the same ISA, not even between different GPU models from the same manufacturer. Although most modern GPUs have converged to SIMD based architectures, the ISA is still very different between manufacturers and it still changes from generation to generation (sometimes different chips of the same generation have slightly different ISA). GPU makers keep adding new instructions when they identify new ways to implement some features more effectively. To deal with all that mess, graphics drivers have to do online compilation of shaders (as opposed to offline compilation which usually happens for apps running on your CPU). This means that shaders have to be recompiled when the

userspace graphics driver is updated either because new functionality is available or because bug fixes were added to the driver and/or compiler. But I only downloaded one driver! On some systems (especially proprietary operating systems like Windows), GPU manufacturers intend to make users' lives easier by offering all of the above in a single installer package, which is just called "the driver". Typically such a package includes: Firmware files for all hardware that the package supports A kernel driver Several userspace drivers for various APIs A shader compiler (sometimes more) that is used by those userspace drivers A "user-friendly" application (ie. a control panel) to present all the functionality to the user Various other utilities and libraries (that you may or may not need). But I didn't download any drivers! On some systems (typically on open source systems like Linux distributions), usually you can already find a set of packages to handle most common hardware, so you can use most functionality out of the box without needing to install anything manually. Neat, isn't it? However, on open source systems, the graphics stack is more transparent, which means that there are many parts that are scattered across different projects, and in some cases there is more than one driver available for the same HW. To end users, it can be very confusing. However, this doesn't mean that open source drivers are designed worse. It is just that due to their community oriented nature, they are organized differently. One of the main sources of confusion is that various Linux distributions mix and match different versions of the kernel with different versions of different UMDs which means that users of different distros can get a wildly different user experience based on the choices made for them by the developers of the distro. Another source of confusion is that we driver developers are really, really bad at naming things, so sometimes different projects end up having the same name, or some projects have nonsensical or outdated names. The Linux graphics stack In the next post, I'll continue this story and discuss how the above applies to the open source Linux graphics stack.

- [Hari Rana: Please Fund My Continued Accessibility Work on GNOME!](#) (2025/12/16 00:00)

Hey, I have been under distress lately due to personal circumstances that are outside my control. I cannot find a permanent job that allows me to function, I am not eligible for government benefits, my grant proposals to work on free and open-source projects got rejected, paid internships are quite difficult to find, especially when many of them prioritize new contributors. Essentially, I have no stable, monthly income that allows me to sustain myself. Nowadays, I mostly volunteer to improve accessibility throughout GNOME apps, either by enhancing the user experience for people with disabilities, or enabling them to use them. I helped make most of GNOME Calendar accessible with a keyboard and screen reader, with additional ongoing effort involving merge requests !564 and !598 to make the month view accessible, all of which is an effort no company has ever contributed to, or would ever contribute to financially. These merge requests require literal thousands of hours for research, development, and testing, enough to sustain me for several years if I were employed. I would really appreciate any kinds of donations, especially ones that happen periodically to increase my monthly income. These donations will allow me to sustain myself while allowing me to work on accessibility throughout GNOME, essentially 'crowdfunding' development without doing it on the behalf of the GNOME Foundation or another organization. Donate on Liberapay Support on Ko-fi Sponsor on GitHub Send via PayPal

- [Sebastian Wick: Flatpak Pre-Installation Approaches](#) (2025/12/13 17:17)

Together with my then-colleague Kalev Lember, I recently added support for pre-installing Flatpak applications. It sounds fancy, but it is conceptually very simple: Flatpak reads configuration files from several directories to determine which applications should be pre-installed. It then installs any missing applications and removes any that are no longer supposed to be pre-installed (with some small caveats). For example, the following configuration tells Flatpak that the devel branch of the app org.test.Foo from remotes which serve the collection org.test.Collection, and the app org.test.Bar from any remote should be installed: [Flatpak Preinstall org.test.Foo] CollectionID=org.test.Collection Branch=devel

[Flatpak Preinstall org.test.Bar] By dropping in another configuration file with a higher priority, pre-installation of the app org.test.Foo can be disabled: [Flatpak Preinstall org.test.Foo] Install=false The installation procedure is the same as it is for the flatpak-install command. It supports installing from remotes and from side-load repositories, which is to say from a repository on a filesystem. This simplicity also means that system integrators are responsible for assembling all the parts into a functioning system, and that there are a number of choices that need to be made for installation and upgrades. The simplest way to approach this is to just ship a bunch of config files in /usr/share/flatpak/preinstall.d and config files for the remotes from which the apps are available. In the installation procedure, flatpak-preinstall is called and it will download the Flatpaks from the remotes over the network into /var/lib/flatpak. This works just fine, until someone needs one of those apps but doesn't have a suitable network connection. The next way one could approach this is exactly the same way, but with a sideload repository on the installation medium which contains the apps that will get pre-installed. The flatpak-preinstall command needs to be pointed at this repository at install time, and the process which creates the installation medium needs to be adjusted to create this repository. The installation process now works without a network connection. System updates are usually downloaded over the network, just as new pre-installed applications will be. It is also possible to simply skip flatpak-preinstall, and use flatpak-install to create a Flatpak installation containing the pre-installed apps which get shipped on the installation medium. This installation can then be copied over from the installation medium to /var/lib/flatpak in the installation process. It unfortunately also makes the installation process less flexible because it becomes impossible to dynamically build the configuration. On modern, image-based operating systems, it might be tempting to just ship this Flatpak installation on the image because the flexibility is usually neither required nor wanted. This currently does not work for the simple reason that the default system installation is in /var/lib/flatpak, which is not in /usr which is the mount point of the image. If the default system installation was in the image, then it would be read-only because the image is read-only. This means we could not update or install anything new to the system installation. If we make it possible to have two different system installations — one in the image, and one in /var — then we could update and install new things, but the installation on the image would become useless over time because all the runtimes and apps will be in /var anyway as they get updated. All of those issues mean that even for image-based operating systems, pre-installation via a sideload repository is not a bad idea for now. It is however also not perfect. The kind of "pure" installation medium which is simply an image now contains a sideload repository. It also means that a factory reset functionality is not possible because the image does not contain the pre-installed apps. In the future, we will need to revisit these approaches to find a solution that works seamlessly with image-based operating systems and supports factory reset functionality. Until then, we can use the systems mentioned above to start rolling out pre-installed Flatpaks.

- [Dave Airlie \(blogspot\): fedora 43: bad mesa update oopsie](#) (2025/11/24 01:42)

F43 picked up the two patches I created to fix a bunch of deadlocks on laptops reported in my previous blog posting. Turns out Vulkan layers have a subtle thing I missed, and I removed a line from the device select layer that would only matter if you have another layer, which happens under steam. The fedora update process caught this, but it still got published which was a mistake, need to probably give changes like this more karma thresholds. I've released a new update <https://bodhi.fedoraproject.org/updates/FEDORA-2025-2f4ba7cd17> that hopefully fixes this. I'll keep an eye on the karma.

- [Juan A. Suarez: Major Upgrades to the Raspberry Pi GPU Driver Stack \(XDC 2025 Recap\)](#) (2025/11/23 23:00)

XDC 2025 happened at the end of September, beginning of October this year, in Kuppelsaal, the historic TU Wien building in Vienna. XDC, The X.Org Developer's Conference, is truly the premier gathering for open-source graphics development. The atmosphere was, as always, highly

collaborative and packed with experts across the entire stack. I was thrilled to present, together with my workmate Ella Stanforth, on the progress we have made in enhancing the Raspberry Pi GPU driver stack. Representing the broader Igalia Graphics Team that work on this GPU, Ella and I detailed the strides we have made in the OpenGL driver, though part of the improvements affect also the Vulkan driver. The presentation was divided in two parts. In the first one, we talked about the new features that we were implementing, or are under implementation, mainly to make the driver more closely aligned with OpenGL 3.2. Key features explained were 16-bit Normalized Format support, Robust Context support, and Seamless cubemap implementation. Beyond these core OpenGL updates, we also highlighted other features, such as NIR printf support, framebuffer fetch or dual source blend, which is important for some game emulators. The second part was focused on specific work done to improve the performance. Here, we started with different traces from the popular GFXBench application, and explained the main improvements done throughout the year, with a look at how much each of these changes improved the performance for each of the benchmarks (or in average). At the end, for some benchmarks we nearly doubled the performance compared to last year. I won't explain here each of the changes done, But I encourage the reader to watch the talk, which is already available. For those that prefer to check the slides instead of the full video, you can view them here: Outside of the technical track, the venue's location provided some excellent down time opportunities to have lunch at different nearby places. I need to highlight here one that I really enjoyed: An's Kitchen Karlsplatz. This cozy Vietnamese street food spot quickly became one of my favourite places, and I went there a couple of times. On the last day, I also had the opportunity to visit some of the most recomendable sightseeings spots in Vienna. Of course, one needs more than a half-day to do a proper visit, but at least it helps to spark an interest to write it down to pay a full visit to the city. Meanwhile, I would like to thank all the conference organizers, as well as all the attendees, and I look forward to see them again.

- [Lennart Poettering: Mastodon Stories for systemd v258](#) (2025/11/17 23:00)

Already on Sep 17 we released systemd v258 into the wild. In the weeks leading up to that release I have posted a series of serieses of posts to Mastodon about key new features in this release, under the #systemd258 hash tag. It was my intention to post a link list here on this blog right after completing that series, but I simply forgot! Hence, in case you aren't using Mastodon, but would like to read up, here's a list of all 37 posts: Post #1: `systemctl start -v` Post #2: Home areas Post #3: `systemd-resolved` delegate zones Post #4: Foreign UID range Post #5: `/etc/hostname` ??? wildcards Post #6: Quota on `/tmp/` Post #7: `ConcurrentnySoftMax=` + `ConcurrencyHardMax=` Post #8: Product UUID in `ConditionHost=` Post #9: Context OSC terminal sequences Post #10: `uki-url` Boot Loader Spec Type #1 fields Post #11: `rd.break=` boot breakpoints Post #12: Factory Reset Rework Post #13: `systemd-resolved` DNS Configuration Change IPC Subscription API Post #14: `io.systemd.boot-entries.extra=` SMBIOS Type #11 Key Post #15: Bring Your Own Firmware Post #16: `userdb` record aliases Post #17: `systemd-validatefs` and its `xattrs` Post #18: Offline Signing of Artifacts Post #19: `PAMName=` in services hooked up to `ask-password` protocol Post #20: `x-systemd.graceful-option=` mount option Post #21: `systemd-userdb-load-credentials.service` Post #22: `systemd-vmspawn --grow-image=a` Post #23: `systemd-notify --fork` Post #24: `$TERM` auto-discovery Post #25: Rebooting/Powering off `systemd-nspawn` containers via hotkey Post #26: `ExecStart=` | modifier Post #27: `systemctl reload reloads confexts` Post #28: Server side `userdb` filtering Post #29: Quota on `StateDirectory=` and friends Post #30: `systemd-analyze unit-shell` Post #31: `/etc/issue.d/` drop-in for `AF_VSOCK` CID Post #32: `fsverity` in `systemd-repart` Post #33: `AcceptFileDescriptor=` + `PassPIDFD=` Post #34: Tab completion in interactive `systemd-firstboot` Post #35: `rd.systemd.pull=` kernel command line option/Boot into tarball Post #36: `ConditionKernelModuleLoaded=` Post #37: `systemd-analyze chid` Post #38: `homectl list-signing-keys/get-signing-key/add-signing-key/remove-signing-key` Post #39: DDI Image Filters Post #40: Android USB Debugging `udev` rules Post #41: `systemd-vmspawn's --smbios11=`

switch Post #42: \$MAINPIDFDID + \$MANAGERPIDFDID Post #43: \$DEBUG_INVOCATION=1 Respected by all systemd services Post #44: LoaderDeviceURL EFI Variable and systemd.pull='s origin kernel command line switch Post #45: cgroupv1 removal Post #46: ProtectHostname=private Post #47: homectl adopt + homectl register Post #48: systemd-machined Varlink APIs Post #49: DeferTrigger and "lenient" job mode Post #50: Automatic Removal of foreign UID owned delegate subgroups in the per-user service manager Post #51: Per-user ask-password protocol Post #52: PrivateUsers=full Post #53: LoadCredentialEncrypted= in the per-user service manager Post #54: dissect_image builtin in systemd-udev Post #55: BPF Delegation via Tokens I intend to do a similar series of serieses of posts for the next systemd release (v259), hence if you haven't left tech Twitter for Mastodon yet, now is the opportunity. We intend to shorten the release cycle a bit for the future, and in fact managed to tag v259-rc1 already yesterday, just 2 months after v258. Hence, my series for v259 will begin soon, under the #systemd259 hash tag. In case you are interested, here is the corresponding blog story for systemd v257, and here for v256.

- [Rodrigo Siqueira: XDC 2025](#) (2025/11/17 00:00)

It has been a long time since I published any update in this space. Since this was a year of colossal changes for me, maybe it is also time for me to make something different with this blog and publish something just for a change — why not start talking about XDC 2025? This year, I attended XDC 2025 in Vienna as an Igalia developer. I was thrilled to see some faces from people I worked with in the past and people I'm working with now. I had a chance to hang out with some folks I worked with at AMD (Harry, Alex, Leo, Christian, Shashank, and Pierre), many Igalians (Žan, Job, Ricardo, Paulo, Tvrtko, and many others), and finally some developers from Valve. In particular, I met Tímur in person for the first time, even though we have been talking for months about GPU recovery. Speaking of GPU recovery, we held a workshop on this topic together. The workshop was packed with developers from different companies, which was nice because it added different angles on this topic. We began our discussion by focusing on the topic of job resubmission. Christian began sharing a brief history of how the AMDGPU driver started handling resubmission and the associated issues. After learning from firsthand experience, amdgpu ended up adopting the following approach: When a job cause a hang, call driver specific handler. Stop the scheduler. Copy all jobs from the ring buffer, minus the job that caused the issue, to a temporary ring. Reset the ring buffer. Copy back the other jobs to the ring buffer. Resume the scheduler. Below, you can see one crucial series associated with amdgpu recovery implementation:

<https://lore.kernel.org/amd-gfx/20250701184451.11868-1-alexander.deucher@amd.com/T/#m5df543b5cbbdc31e2834c955d1e3e30b939cb77>

The next topic was a discussion around the replacement of `drm_sched_resubmit_jobs()` since this function became deprecated. Just a few drivers still use this function, and they need a replacement for that. Some ideas were floating around to extract part of the specific implementation from some drivers into a generic function. The next day, Philipp Stanner continued to discuss this topic in his workshop, DRM GPU Scheduler. Another crucial topic discussed was improving GPU reset debuggability to narrow down which operations cause the hang (keep in mind that GPU recovery is a medicine, not the cure to the problem). Intel developers shared their strategy for dealing with this by obtaining hints from userspace, which helped them provide a better set of information to append to the `devcoredump`. AMD could adopt this alongside dumping the IB data into the `devcoredump` (I am already investigating this). Finally, we discussed strategies to avoid hang issues regressions. In summary, we have two lines of defense: IGT: At the IGT level, we can have more tests that insert malicious instructions into the ring buffer, forcing the driver into an invalid state and triggering the recovery process. HangTest suite: HangTest suite is a tool that simulates some potential hangs using Vulkan. Some tests are already available in this suite, but we should explore more creative combinations for trying to trigger hangs. Lighting talk This year, as always, XDC was super cool, packed with many engaging presentations which I highly recommend everyone check out. If you are interested,

check the schedule and the presentation recordings available on the X.Org Foundation Youtube page. Anyway, I hope this blog post marks the inauguration of a new era for this site, where I will start posting more content ranging from updates to tutorials. See you soon.

- [Simon Ser: Status update, November 2025](#) (2025/11/15 22:00)

Hi! This month a lot of new features have added to the Goguma mobile IRC client. Hubert Hirtz has implemented drafts so that unsent text gets saved and network disconnections don't disrupt users typing a message. He also enabled replying to one's own messages, changed the appearance of short messages containing only emoji, upgraded our emoji library to Unicode version 16, fixed some linkifier bugs and added unit tests. Markus Cislser has added a new option in the message menu to show a user's profile. I've added an on-disk cache for images (with our own implementation, because the widely used `cached_network_image` package is heavyweight). I've been working on displaying network icons and blocking users, but that work is not finished yet. I've also contributed some maintenance fixes for our `webcrypto.dart` dependency (toolkit upgrades and CI fixes). The soju IRC bouncer has also got some love this month. delthas has contributed support for labeled-response for soju clients, allowing more reliable matching of server replies with client commands. I've introduced a new icon directive to configure an image representing the bouncer. soju v0.10.0 has been released, followed by soju v0.10.1 including bug fixes from Karel Balej and Taavi Väänänen. In Wayland news, `wlroots` v0.19.2 and v0.18.3 have been released thanks to Simon Zeni. I've added support for the color-representation protocol for the Vulkan renderer, allowing clients to configure the color encoding and range for YCbCr content. Félix Poisot has been hard at work with more color management patches: screen default color primaries are now extracted from the EDID and exposed to compositors, the cursor is now correctly converted to the output's primaries and transfer function, and some work-in-progress patches switch the renderer API from a descriptive model to a prescriptive model. `go-webdav` v0.7.0 has been released with a patch from prasad83 to play well with Thunderbird. I've updated clients to make multi-status errors non-fatal, returning partial data alongside the error. I've released `drm_info` v2.9.0 with improvements mentioned in the previous status update plus support for the TILE connector property. See you next month!

- [Dave Airlie \(blogspot\): a tale of vulkan/nouveau/nvk/zink/mutter + deadlocks](#) (2025/11/10 03:16)

I had a bug appear in my email recently which led me down a rabbit hole, and I'm going to share it for future people wondering why we can't have nice things. Bug: 1. Get an intel/nvidia (newer than Turing) laptop. 2. Log in to GNOME on Fedora 42/43 3. Hotplug a HDMI port that is connected to the NVIDIA GPU. 4. Desktop stops working. My initial reproduction got me a hung mutter process with a nice backtrace which pointed at the Vulkan Mesa device selection layer, trying to talk to the wayland compositor to ask it what the default device is. The problem was the process was the wayland compositor, and how was this ever supposed to work. The Vulkan device selection was called because zink called `EnumeratePhysicalDevices`, and zink was being loaded because we recently switched to it as the OpenGL driver for newer NVIDIA GPUs. I looked into zink and the device select layer code, and lo and behold someone has hacked around this badly already, and probably wrongly and I've no idea what the code does, because I think there is at least one logic bug in it. Nice things can't be had because hacks were done instead of just solving the problem. The hacks in place ensured under certain circumstances involving zink/xwayland that the device select code to probe the window system was disabled, due to deadlocks seen. I'd no idea if more hacks were going to help, so I decided to step back and try and work out better. The first question I had is why `WAYLAND_DISPLAY` is set inside the compositor process, it is, and if it wasn't I would never hit this. It's pretty likely on the initial compositor start this env var isn't set, so the problem only becomes apparent when the compositor gets a hotplugged GPU output, and goes to load the OpenGL driver, zink, which enumerates and hits device select with env var set and deadlocks. I wasn't going to figure out a way around `WAYLAND_DISPLAY` being set at this point, so I leave the above question as an exercise for mutter devs. How do I fix

it?Attempt 1:At the point where zink is loading in mesa for this case, we have the file descriptor of the GPU device that we want to load a driver for. We don't actually need to enumerate all the physical devices, we could just find the ones for that fd. There is no API for this in Vulkan. I wrote an initial proof of concept instance extensions call `VK_MESA_enumerate_devices_fd`. I wrote initial loader code to play with it, and wrote zink code to use it. Because this is a new instance API, device-select will also ignore it. However this ran into a big problem in the Vulkan loader. The loader is designed around some internals that `PhysicalDevices` will enumerate in similar ways, and it has to trampoline `PhysicalDevice` handles to underlying driver pointers so that if an app enumerates once, and enumerates again later, the `PhysicalDevice` handles remain consistent for the first user. There is a lot of code, and I've no idea how hotplug GPUs might fail in such situations. I couldn't find a decent path forward without knowing a lot more about the Vulkan loader. I believe this is the proper solution, as we know the fd, we should be able to get things without doing a full enumeration then picking the answer using the fd info. I've asked Vulkan WG to take a look at this, but I still need to fix the bug.

Attempt 2:Maybe I can just turn off device selection, like the current hacks do, but in a better manner. Enter `VK_EXT_layer_settings`. This extensions allows layers to expose a layer setting in the instance creation. I can have the device select layer expose a setting which says don't touch this instance. Then in the zink code where we have a file descriptor being passed in and create an instance, we set the layer setting to avoid device selection. This seems to work but it has some caveats, I need to consider, but I think should be fine.

zink uses a single `VkInstance` for it's device screen. This is shared between all `pipe_screens`. Now I think this is fine inside a compositor, since we shouldn't ever be loading zink via the non-fd path, and I hope for most use cases it will work fine, better than the current hacks and better than some other ideas we threw around. The code for this is in [1].

What else might be affected:If you have a vulkan compositor, it might be worth setting the layer setting if the mesa device select layer is loaded, esp if you set the `DISPLAY/WAYLAND_DISPLAY` and do any sort of hotplug later. You might be safe if you Enumerate`PhysicalDevices` early enough, the reason it's a big problem in mutter is it doesn't use Vulkan, it uses OpenGL and we only enumerate Vulkan physical devices at runtime through zink, never at startup.

AMD and NVIDIA I think have proprietary device selection layers, these might also deadlock in similar ways, I think we've seen some wierd deadlocks in NVIDIA driver enumerations as well that might be a similar problem. [1] https://gitlab.freedesktop.org/mesa/mesa/-/merge_requests/38252

- [Sebastian Wick: Flatpak Happenings](#) (2025/11/04 20:28)

Yesterday I released Flatpak 1.17.0. It is the first version of the unstable 1.17 series and the first release in 6 months. There are a few things which didn't make it for this release, which is why I'm planning to do another unstable release rather soon, and then a stable release still this year. Back at LAS this year I talked about the Future of Flatpak and I started with the grim situation the project found itself in: Flatpak was stagnant, the maintainers left the project and PRs didn't get reviewed. Some good news: things are a bit better now. I have taken over maintenance, Alex Larsson and Owen Taylor managed to set aside enough time to make this happen and Boudhayan Bhattcharya (bbhtt) and Adrian Vovk also got more involved. The backlog has been reduced considerably and new PRs get reviewed in a reasonable time frame. I also listed a number of improvements that we had planned, and we made progress on most of them: It is now possible to define which Flatpak apps shall be pre-installed on a system, and Flatpak will automatically install and uninstall things accordingly. Our friends at Aurora and Bluefin already use this to ship core apps from Flathub on their bootc based systems (shout-out to Jorge Castro). The OCI support in Flatpak has been enhanced to support pre-installing from OCI images and remotes, which will be used in RHEL 10 We merged the backwards-compatible permission system. This allows apps to use new, more restricting permissions, while not breaking compatibility when the app runs on older systems. Specifically access to input devices such as gamepads, and access to the USB portal can now be granted in this way. It will also help us to transition to

PipeWire. We have up-to-date docs for libflatpak again Besides the changes directly in Flatpak, there are a lot of other things happening around the wider ecosystem: bbhttp released a new version of flatpak-builder Enhanced License Compliance Tools for Flathub Adrian and I have made plans for a service which allows querying running app instances (systemd-appd). This provides a new way of authenticating Flatpak instances and is a prerequisite for nested sandboxing, PipeWire support, and getting rid of the D-Bus proxy. My previous blog post went into a few more details. Our friends at KDE have started looking into the XDG Intents spec, which will hopefully allow us to implement deep-linking, thumbnailing in Flatpak apps, and other interesting features Adrian made progress on the session save/restore Portal Some rather big refactoring work in the Portals frontend, and GDBus and libdex integration work which will reduce the complexity of asynchronous D-Bus What I have also talked about at my LAS talk is the idea of a Flatpak-Next project. People got excited about this, but I feel like I have to make something very clear: If we redid Flatpak now, it would not be significantly better than the current Flatpak! You could still not do nested sandboxing, you would still need a D-Bus proxy, you would still have a complex permission system, and so on. Those problems require work outside of Flatpak, but have to integrate with Flatpak and Flatpak-Next in the future. Some of the things we will be doing include: Work on the systemd-appd concept Make varlink a feasible alternative to D-Bus D-Bus filtering in the D-Bus daemons Network sandboxing via pasta PipeWire policy for sandboxes New Portals So if you're excited about Flatpak-Next, help us to improve the Flatpak ecosystem and make Flatpak-Next more feasible!

- [Melissa Wen: Kworkflow at Kernel Recipes 2025](#) (2025/11/03 21:30)

This was the first year I attended Kernel Recipes and I have nothing but say how much I enjoyed it and how grateful I'm for the opportunity to talk more about kworkflow to very experienced kernel developers. What I mostly like about Kernel Recipes is its intimate format, with only one track and many moments to get closer to experts and people that you commonly talk online during your whole year. In the beginning of this year, I gave the talk Don't let your motivation go, save time with kworkflow at FOSDEM, introducing kworkflow to a more diversified audience, with different levels of involvement in the Linux kernel development. At this year's Kernel Recipes I presented the second talk of the first day: Kworkflow - mix & match kernel recipes end-to-end. The Kernel Recipes audience is a bit different from FOSDEM, with mostly long-term kernel developers, so I decided to just go directly to the point. I showed kworkflow being part of the daily life of a typical kernel developer from the local setup to install a custom kernel in different target machines to the point of sending and applying patches to/from the mailing list. In short, I showed how to mix and match kernel workflow recipes end-to-end. As I was a bit fast when showing some features during my presentation, in this blog post I explain each slide from my speaker notes. You can see a summary of this presentation in the Kernel Recipe Live Blog Day 1: morning. Introduction Hi, I'm Melissa Wen from Igalia. As we already started sharing kernel recipes and even more is coming in the next three days, in this presentation I'll talk about kworkflow: a cookbook to mix & match kernel recipes end-to-end. This is my first time attending Kernel Recipes, so lemme introduce myself briefly. As I said, I work for Igalia, I work mostly on kernel GPU drivers in the DRM subsystem. In the past, I co-maintained VKMS and the v3d driver. Nowadays I focus on the AMD display driver, mostly for the Steam Deck. Besides code, I contribute to the Linux kernel by mentoring several newcomers in Outreachy, Google Summer of Code and Igalia Coding Experience. Also, by documenting and tooling the kernel. And what's this cookbook called kworkflow? Kworkflow (kw) Kworkflow is a tool created by Rodrigo Siqueira, my colleague at Igalia. It's a single platform that combines software and tools to: optimize your kernel development workflow; reduce time spent in repetitive tasks; standardize best practices; ensure that deployment data flows smoothly and reliably between different kernel workflows; It's mostly done by volunteers, kernel developers using their spare time. Its features cover real use cases according to kernel developer needs. Basically it's mixing and matching the daily life of a typical kernel developer with kernel workflow recipes with some secret sauces. First recipe: A good GPU

driver for my AMD laptop So, it's time to start the first recipe: A good GPU driver for my AMD laptop. Before starting any recipe we need to check the necessary ingredients and tools. So, let's check what you have at home. With kworkflow, you can use: kw device: to get information about the target machine, such as: CPU model, kernel version, distribution, GPU model, kw remote: to set the address of this machine for remote access kw config: you can configure kw with kw config. With this command you can basically select the tools, flags and preferences that kw will use to build and deploy a custom kernel in a target machine. You can also define recipients of your patches when sending it using kw send-patch. I'll explain more about each feature later in this presentation. kw kernel-config manager (or just kw k): to fetch the kernel .config file from a given machine, store multiple .config files, list and retrieve them according to your needs. Now, with all ingredients and tools selected and well portioned, follow the right steps to prepare your custom kernel! First step: Mix ingredients with kw build or just kw b kw b and its options wrap many routines of compiling a custom kernel. You can run kw b -i to check the name and kernel version and the number of modules that will be compiled and kw b --menu to change kernel configurations. You can also pre-configure compiling preferences in kw config regarding kernel building. For example, target architecture, the name of the generated kernel image, if you need to cross-compile this kernel for a different system and which tool to use for it, setting different warning levels, compiling with CFlags, etc. Then you can just run kw b to compile the custom kernel for a target machine. Second step: Bake it with kw deploy or just kw d After compiling the custom kernel, we want to install it in the target machine. Check the name of the custom kernel built: 6.17.0-rc6 and with kw s SSH access the target machine and see it's running the kernel from the Debian distribution 6.16.7+deb14-amd64. As with building settings, you can also pre-configure some deployment settings, such as compression type, path to device tree binaries, target machine (remote, local, vm), if you want to reboot the target machine just after deploying your custom kernel, and if you want to boot in the custom kernel when restarting the system after deployment. If you didn't pre-configure some options, you can still customize as a command option, for example: kw d --reboot will reboot the system after deployment, even if I didn't set this in my preference. With just running kw d --reboot I have installed the kernel in a given target machine and rebooted it. So when accessing the system again I can see it was booted in my custom kernel. Third step: Time to taste with kw debug kw debug wraps many tools for validating a kernel in a target machine. We can log basic dmesg info but also tracking events and ftrace. With kw debug --dmesg --history we can grab the full dmesg log from a remote machine, if you use the --follow option, you will monitor dmesg outputs. You can also run a command with kw debug --dmesg --cmd="<my command>" and just collect the dmesg output related to this specific execution period. In the example, I'll just unload the amdgpu driver. I use kw drm --gui-off to drop the graphical interface and release the amdgpu for unloading it. So I run kw debug --dmesg --cmd="modprobe -r amdgpu" to unload the amdgpu driver, but it fails and I couldn't unload it. Cooking Problems Oh no! That custom kernel isn't tasting good. Don't worry, as in many recipes preparations, we can search on the internet to find suggestions on how to make it tasteful, alternative ingredients and other flavours according to your taste. With kw patch-hub you can search on the lore kernel mailing list for possible patches that can fix your kernel issue. You can navigate in the mailing lists, check series, bookmark it if you find it relevant and apply it in your local kernel tree, creating a different branch for tasting... oops, for testing. In this example, I'm opening the amd-gfx mailing list where I can find contributions related to the AMD GPU driver, bookmark and/or just apply the series to my work tree and with kw bd I can compile & install the custom kernel with this possible bug fix in one shot. As I changed my kw config to reboot after deployment, I just need to wait for the system to boot to try again unloading the amdgpu driver with kw debug --dmesg --cm=modprobe -r amdgpu. From the dmesg output retrieved by kw for this command, the driver was unloaded, the problem is fixed by this series and the kernel tastes good now. If I'm satisfied with the solution, I can even use kw patch-hub to access the bookmarked series and marking the checkbox that will reply the patch thread with a

Reviewed-by tag for me. Second Recipe: Raspberry Pi 4 with Upstream Kernel As in all recipes, we need ingredients and tools, but with kworkflow you can get everything set as when changing scenarios in a TV show. We can use kw env to change to a different environment with all kw and kernel configuration set and also with the latest compiled kernel cached. I was preparing the first recipe for a x86 AMD laptop and with kw env --use RPI_64 I use the same worktree but moved to a different kernel workflow, now for Raspberry Pi 4 64 bits. The previous compiled kernel 6.17.0-rc6-mainline+ is there with 1266 modules, not the 6.17.0-rc6 kernel with 285 modules that I just built&deployed. kw build settings are also different, now I'm targeting a arm64 architecture with a cross-compiled kernel using aarch64-linu-gnu- cross-compilation tool and my kernel image calls kernel8 now. If you didn't plan for this recipe in advance, don't worry. You can create a new environment with kw env --create RPI_64_V2 and run kw init --template to start preparing your kernel recipe with the mirepoix ready. I mean, with the basic ingredients already cut... I mean, with the kw configuration set from a template. And you can use kw remote to set the IP address of your target machine and kw kernel-config-manager to fetch/retrieve the .config file from your target machine. So just run kw bd to compile and install a upstream kernel for Raspberry Pi 4. Third Recipe: The Mainline Kernel Ringing on my Steam Deck (Live Demo) Let's show you how easy is to build, install and test a custom kernel for Steam Deck with Kworkflow. It's a live demo, but I also recorded it because I know the risks I'm exposed to and something can go very wrong just because of reasons :) Report: how was the live demo For this live demo, I took my OLED Steam Deck to the stage. I explained that, if I boot mainline kernel on this device, there is no audio. So I turned it on and booted the mainline kernel I've installed beforehand. It was clear that there was no typical Steam Deck startup audio when the system was loaded. As I started the demo in the kw environment for Raspberry Pi 4, I first moved to another environment previously used for Steam Deck. In this STEAMDECK environment, the mainline kernel was already compiled and cached, and all settings for accessing the target machine, compiling and installing a custom kernel were retrieved automatically. My live demo followed these steps: With kw env --use STEAMDECK, switch to a kworkflow environment for Steam Deck kernel development. With kw b -i, shows that kw will compile and install a kernel with 285 modules named 6.17.0-rc6-mainline-for-deck. Run kw config to show that, in this environment, kw configuration changes to x86 architecture and without cross-compilation. Run kw device to display information about the Steam Deck device, i.e. the target machine. It also proves that the remote access - user and IP - for this Steam Deck was already configured when using the STEAMDECK environment, as expected. Using git am, as usual, apply a hot fix on top of the mainline kernel. This hot fix makes the audio play again on Steam Deck. With kw b, build the kernel with the audio change. It will be fast because we are only compiling the affected files since everything was previously done and cached. Compiled kernel, kw configuration and kernel configuration is retrieved by just moving to the "STEAMDECK" environment. Run kw d --force --reboot to deploy the new custom kernel to the target machine. The --force option enables us to install the mainline kernel even if mkinitcpio complains about missing support for downstream packages when generating initramfs. The --reboot option makes the device reboot the Steam Deck automatically, just after the deployment completion. After finishing deployment, the Steam Deck will reboot on the new custom kernel version and made a clear resonant or vibrating sound. [Hopefully] Finally, I showed to the audience that, if I wanted to send this patch upstream, I just needed to run kw send-patch and kw would automatically add subsystem maintainers, reviewers and mailing lists for the affected files as recipients, and send the patch to the upstream community assessment. As I didn't want to create unnecessary noise, I just did a dry-run with kw send-patch -s --simulate to explain how it looks. What else can kworkflow already mix & match? In this presentation, I showed that kworkflow supported different kernel development workflows, i.e., multiple distributions, different bootloaders and architectures, different target machines, different debugging tools and automatize your kernel development routines best practices, from development environment setup and verifying a custom kernel in bare-metal to sending contributions

upstream following the contributions-by-e-mail structure. I exemplified it with three different target machines: my ordinary x86 AMD laptop with Debian, Raspberry Pi 4 with arm64 Raspbian (cross-compilation) and the Steam Deck with SteamOS (x86 Arch-based OS). Besides those distributions, Kworkflow also supports Ubuntu, Fedora and PopOS. Now it's your turn: Do you have any secret recipes to share? Please share with us via kworkflow. Useful links Talk Recording of Kworkflow at Kernel Recipes 2025 on Igalia's Channel Talk Abstract, Recording and Slide Deck of Kworkflow at Kernel Recipes 2025 on Kernel Recipes Website Talk Slide Deck for Download with some Videos instead of GIFs

- [Mike Blumenkrantz: Hibernate On](#) (2025/10/31 00:00)

Take A Break We've reached Q4 of another year, and after the mad scramble that has been crunch-time over the past few weeks, it's time for SGC to once again retire into a deep, restful sleep. 2025 saw a lot of ground covered: NVK-Zink synergy Continued Rusticl improvements Viewperf perf and general CPU overhead reduction Tiler GPU perf Mesh shaders apitrace perf More GL extensions released than any other year this decade It's been a real roller coaster ride of a year as always, but I can say authoritatively that fans of the blog, you need to take care of yourselves. You need to use this break time wisely. Rest. Recover. Train your bodies. Travel and broaden your horizons. Invest in night classes to expand your minds. You are not prepared for the insanity that will be this blog in 2026.

- [Mike Blumenkrantz: Apitrace Goes Vroom](#) (2025/10/27 00:00)

First Time Today marks the first post of a type that I've wanted to have for a long while: a guest post. There are lots of graphics developers who work on cool stuff and don't want to waste time setting up blogs, but with enough cajoling they will write a single blog post. If you're out there thinking you just did some awesome work and you want the world to know the grimy, gory details, let me know. The first victimrecipient of this honor is an individual famous for small and extremely sane endeavors such as descriptor buffers in Lavapipe, ray tracing in Lavapipe, and sparse support in Lavapipe. Also wrangling ray tracing for RADV. Below is the debut blog post by none other than Konstantin Seurer. What is apitrace? Apitrace is a powerful tool for capturing and replaying traces of GL and DX applications. The problem is that it is not really suitable for performance testing. This blog post is about implementing a faster method for replaying traces. About six weeks ago, Mike asked me if I wanted to work on this. [6:58:58 pm] <zmike> on the topic of traces [6:59:08 pm] <zmike> I have a longer-term project that could use your expertise [6:59:19 pm] <zmike> it's low work but high complexity [7:00:12 pm] <zmike> specifically I would like apitrace to be able to competently output C code from traces and to have this functionality merged upstream low work Sure. The state of glretrace This first obvious step was measuring how glretrace currently performs. Mike kindly provided a couple of traces from his personal collection, and I immediately timed a trace of the only relevant OpenGL game: \$ time ./glretrace -b minecraft-perf.trace /Users/Cortex/Downloads/graalvm-jdk-23.0.1+11.1/bin/java Rendered 1261 frames in 10.4269 secs, average of 120.937 fps real 0m10.554s user 0m12.938s sys 0m2.712s This looks fine, but I have no idea how fast the application is supposed to run. Running the same trace with perf reveals that there is room for improvement. 2/3 of frametime is spent parsing the trace. Implementation An apitrace trace stores API call information in an object-oriented style. This makes basic codegen really easy because the objects map directly to the generated C/C++ code. However, not all API calls are made equal, and the countless special cases that I needed to handle are what made this project take so long. glretrace has custom implementations for WSI API calls, and it would be a shame not to use them. The easiest way of doing that is generating a shared library instead of an executable and having glretrace load it. The shared library can then provide a bunch of callbacks for the call sequences we can do codegen for and Call objects for everything else. Besides WSI, there are also arguments and return values that need special treatment. OpenGL allows the application to create all kinds of objects that are represented using IDs. Those IDs are assigned by the driver, and they can be different during replay. glretrace remaps them using std::maps which have non-trivial

overhead. I initially did that as well for the codegen to get things up and running, but it is actually possible to emit global variables and have most of the remapping run logic during codegen. Data streaming With the main replay overhead being taken care of, a major amount of replay time is now spent loading texture and buffer data. In large traces, there can also be >10GiB of data, so loading everything upfront is not an option. I decided to create one thread for reading the data file and nproc decompression threads. The read thread will wait if enough data has been loaded to limit memory usage. Decompression threads are needed because decompression is slower than reading the compressed data. codegen in action The results speak for themselves: `$./glretrace --generate-c minecraft-perf minecraft-perf.trace`
`/Users/Cortex/Downloads/graalvm-jdk-23.0.1+11.1/bin/java` Rendered 0 frames in 79.4072 secs, average of 0 fps `$ cd minecraft-perf $ echo "Invoke the superior build tool" $ meson build --buildtype release $ ninja -Cbuild $ time ./glretrace build/minecraft-perf.so` info: Opening 'minecraft-perf.so'... (0.00668795 secs) warning: Waited 0.0461142 secs for data (sequence = 19) Rendered 1261 frames in 5.19587 secs, average of 242.693 fps real 0m5.415s user 0m5.429s sys 0m4.983s Nice. Looking at perf most CPU time is now spent in driver code or streaming binary data for stuff like textures on a separate thread. If you are interested in trying this out yourself, feel free to build the upstream PR and report on bugs unintended features. It would also be nice to have DX support in the future, but that will be something for the dxvk developers unless I need something to procrastinate from doing RT work. - Konstantin

- [Simon Ser: Status update, October 2025](#) (2025/10/15 22:00)

Hi! I skipped last month's status update because I hadn't collected a lot of interesting updates and I've dedicated my time to writing an announcement for the first vali release. Earlier this month, I've taken the train to Vienna to attend XDC 2025. The conference was great, I really enjoyed discussing face-to-face with open-source graphics folks I usually only interact with online, and meeting new awesome people! Since I'm part of the X.Org Foundation board, it was also nice to see the fruit of our efforts. Many thanks to all organizers! We've discussed many interesting topics: a new API for 2D acceleration hardware, adapting the Wayland linux-dmabuf protocol to better support multiple GPUs, some ways to address current Wayback limitations, ideas to improve libliftoff, Vulkan use in Wayland compositors, and a lot more. On the wlroots side, I've worked on a patch to fallback to the renderer to apply gamma LUTs when the KMS driver doesn't support them (this also paves the way for applying color transforms in KMS). Félix Poisot has updated wlroots to support the gamma 2.2 transfer function and use it by default. llyyr has added support for the BT.1886 transfer function and fixed direct scanout for client using the gamma 2.2 transfer function. I've sent a patch to add support for DisplayID v2 CTA-861 data blocks, required for handling some HDR screens. I've reviewed and merged a bunch of gamescope patches to avoid protocol errors with the color management protocol, fix nested mode under a Vulkan compositor, fix a crash on VT switch and modernize dependencies. I've worked a bit on drm_info too. I've added a JSON schema to describe the shape of the JSON objects, made it so EDIDs are included in the JSON output as base64-encoded strings, and added the EDID make/model/serial + bus info to the pretty-printed output. delthas has added soju support for user metadata, introduced a new work-in-progress metadata key to block users, and made it so soju cancels Web Push notifications if a client marks a message as read (to avoid opening notifications for a very short time when actively chatting with another user). Markus Cisler has revamped Goguma's message bubbles: they look much better now! See you next month!

- [Sebastian Wick: SO_PEERPIDFD Gets More Useful](#) (2025/10/10 15:04)

A while ago I wrote about the limited usefulness of SO_PEERPIDFD. for authenticating sandboxed applications. The core problem was simple: while pidfds gave us a race-free way to identify a process, we still had no standardized way to figure out what that process actually was - which sandbox it ran in, what application it represented, or what permissions it should have. The situation has improved considerably since then.

cgroup xattrs Cgroups now support user extended attributes. This feature allows arbitrary metadata to be attached to cgroup inodes using standard xattr calls. We can change flatpak (or snap, or any other container engine) to create a cgroup for application instances it launches, and attach metadata to it using xattrs. This metadata can include the sandboxing engine, application ID, instance ID, and any other information the compositor or D-Bus service might need. Every process belongs to a cgroup, and you can query which cgroup a process belongs to through its pidfd - completely race-free. Standardized Authentication Remember the complexity from the original post? Services had to implement different lookup mechanisms for different sandbox technologies: For flatpak: look in /proc/\$PID/root/.flatpak-info For snap: shell out to snap routine portal-info For firejail: no solution ... All of this goes away. Now there's a single path: Accept a connection on a socket Use SO_PEERPIDFD to get a pidfd for the client Query the client's cgroup using the pidfd Read the cgroup's user xattrs to get the sandbox metadata This works the same way regardless of which sandbox engine launched the application. A Kernel Feature, Not a systemd One It's worth emphasizing: cgroups are a Linux kernel feature. They have no dependency on systemd or any other userspace component. Any process can manage cgroups and attach xattrs to them. The process only needs appropriate permissions and is restricted to a subtree determined by the cgroup namespace it is in. This makes the approach universally applicable across different init systems and distributions. To support non-Linux systems, we might even be able to abstract away the cgroup details, by providing a varlink service to register and query running applications. On Linux, this service would use cgroups and xattrs internally. Replacing Socket-Per-App The old approach - creating dedicated wayland, D-Bus, etc. sockets for each app instance and attaching metadata to the service which gets mapped to connections on that socket - can now be retired. The pidfd + cgroup xattr approach is simpler: one standardized lookup path instead of mounting special sockets. It works everywhere: any service can authenticate any client without special socket setup. And it's more flexible: metadata can be updated after process creation if needed. For compositor and D-Bus service developers, this means you can finally implement proper sandboxed client authentication without needing to understand the internals of every container engine. For sandbox developers, it means you have a standardized way to communicate application identity without implementing custom socket mounting schemes.

- [Mike Blumenkrantz: Mesh Shaders In The Current Year](#) (2025/10/09 00:00)

It Happened. Just a quick post to confirm that the OpenGL/ES Working Group has signed off on the release of GL_EXT_mesh_shader. Credits This is a monumental release, the largest extension shipped for GL this decade, and the culmination of many, many months of work by AMD. In particular we all need to thank Qiang Yu (AMD), who spearheaded this initiative and did the vast majority of the work both in writing the specification and doing the core mesa implementation. Shihao Wang (AMD) took on the difficult task of writing actual CTS cases (not mandatory for EXT extensions in GL, so this is a huge benefit to the ecosystem). Big thanks to both of you, and everyone else behind the scenes at AMD, for making this happen. Also we have to thank the nvidia project and its author, Cortex, for single-handedly pushing the industry forward through the power of Minecraft modding. Stay sane out there. Support Minecraft mod support is already underway, so expect that to happen "soon". The bones of this extension have already been merged into mesa over the past couple months. I opened a MR to enable zink support this morning since I have already merged the implementation. Currently, I'm planning to wait until either just before the branch point next week or until RadeonSI merges its support to merge the zink MR. This is out of respect: Qiang Yu did a huge lift for everyone here, and ideally AMD's driver should be the first to be able to advertise that extension to reflect that. But the branchpoint is coming up in a week, and SGC will be going into hibernation at the end of the month until 2026, so this offer does have an expiration date. In any case, we're done here.

- [Simon Ser: Announcing vali, a C library for Varlink](#) (2025/10/03 22:00)

In the past months I've been working on vali, a C library for Varlink. Today I'm publishing the first vali release! I'd like to explain how to use it for readers who aren't especially familiar with Varlink, and describe some interesting API design decisions. What is Varlink anyways? Varlink is a very simple Remote Procedure Call (RPC) protocol. Clients can call methods exposed by services (ie, servers). To call a method, a client sends a JSON object with its name and parameters over a Unix socket. To reply to a call, a service sends a JSON object with response parameters. That's it. Here's an example request with a bar parameter containing an integer: { "method": "org.example.service.Foo", "parameters": { "bar": 42 } } And here's an example response with a baz parameter containing a list of strings: { "parameters": { "baz": ["hello", "world"] } } Varlink also supports calls with no reply or with multiple replies, but let's leave this out of the picture for simplicity's sake. Varlink services can describe the methods they implement with an interface definition file. method Foo(bar: int) -> (baz: []string) Coming from the Wayland world, I love generating code from specification files. This removes all of the manual encoding/decoding boilerplate and is more type-safe. Unfortunately the official libvarlink library doesn't support code generation (and is not actively maintained anymore), so I've decided to write my own. vali is the result! vali without code generation To better understand the benefits of code generation and vali design decisions, let's take a minute to have a look at what usage without code generation looks like. A client first needs to connect via vali_client_connect_unix(), then call vali_client_call() with a JSON object containing input parameters. It'll get back a JSON object containing output parameters, which needs to be parsed. struct vali_client *client = vali_client_connect_unix("/run/org.example.service"); if (client == NULL) { fprintf(stderr, "Failed to connect to service\n"); exit(1); } struct json_object *in = json_object_new_object(); json_object_object_add(in, "bar", json_object_new_int(42)); struct json_object *out = NULL; if (!vali_client_call(client, "org.example.service.Foo", in, &out, NULL)) { fprintf(stderr, "Foo request failed\n"); exit(1); } struct json_object *baz = json_object_object_get(out, "baz"); for (size_t i = 0; i < json_object_array_length(baz); i++) { struct json_object *item = json_object_array_get_idx(baz, i); printf("%s\n", json_object_get_string(item)); } This is a fair amount of boilerplate. In case of a type mismatch, the client will silently print nothing, which isn't ideal. The last parameter of vali_client_call() is an optional struct vali_error *: if set to a non-NULL pointer and the service replies with an error, the struct is populated, otherwise it's zero'ed out: struct vali_error err; if (!vali_client_call(client, "org.example.service.Foo", in, &out, &err)) { if (err.name != NULL) { fprintf(stderr, "Foo request failed: %s\n", err.name); } else { fprintf(stderr, "Foo request failed: internal error\n"); } vali_error_finish(&err); exit(1); } How does the service side look like? A service first calls vali_service_create() to initialize a fresh service, defines a callback to be invoked when a Varlink call is performed by a client via vali_service_set_call_handler(), and sets up a Unix socket via vali_service_listen_unix(). Let's demonstrate how a service accesses a shared state by printing the number of calls done so far when the callback is invoked. The callback needs to end the call with vali_service_call_close_with_reply(). void handle_call(struct vali_service_call *call, void *user_data) { int *call_count_ptr = user_data; (*call_count_ptr)++; printf("Received %d-th client call\n", *call_count_ptr); struct json_object *baz = json_object_new_array(); json_object_array_add(baz, json_object_new_string("hello")); json_object_array_add(baz, json_object_new_string("world")); struct json_object *params = json_object_new_object(); json_object_object_add(params, "baz", baz); vali_service_call_close_with_reply(call, params); } int main(int argc, void *argv[]) { int call_count = 0; struct vali_service_call_handler handler = { .func = handle_call, .user_data = &call_count, }; struct vali_service *service = vali_service_create(); vali_service_set_call_handler(service, &handler); vali_service_listen_unix(service, "/run/org.example.ftl"); while (vali_service_dispatch(service)); return 0; } In a prior iteration of the API, the callback would return the reply JSON object. This got changed to vali_service_call_close_with_reply() so that services can handle a call asynchronously. If a service needs some time to reply (e.g. because it needs to send data over a network, or perform a long computation), it can give back control to its event loop so that other

clients are not blocked, and later call `vali_service_call_close_with_reply()` from another callback. Why bundle the callback and the user data pointer together in a struct, rather than pass them as two separate parameters to `vali_service_set_call_handler()`? The answer is two-fold: Conceptually, the user data pointer is tied to the callback. Other programming languages with support for lambdas just capture variables. Standard C doesn't have lambdas, and the user data pointer is just a way to pass state to the callback. Bundling the callback and the user data pointer together as a single fat pointer unlocks more ergonomic and safer APIs: a function can return a single struct `vali_service_call_handler` without making the caller manipulate two separate variables to pass it down to `vali_service_set_call_handler()` (and risk mixing them up in case there are multiple). This design makes wrapping a handler much easier (to create middlewares and routers, more on that below). This all might sound familiar to folks who've written an HTTP server: indeed, struct `vali_service_call_handler` is inspired from Go's `net/http.Handler`.

Client side with code generation Given the method definition from the article introduction, `vali` generates the following client function: `struct example_Foo_in { int bar; }; struct example_Foo_out { char **baz; size_t baz_len; }; bool example_Foo(struct vali_client *client, const struct example_Foo_in *in, struct example_Foo *out, struct vali_error *err);` It can be used this way to send the JSON request we've seen earlier: `struct vali_client *client = vali_client_connect_unix("/run/org.example.service"); if (client == NULL) { fprintf(stderr, "Failed to connect to service\n"); exit(1); } const struct example_Foo_in in = { .bar = 42, }; struct example_Foo_out out; if (!example_Foo(client, &in, &out, NULL)) { fprintf(stderr, "Foo request failed\n"); exit(1); } for (size_t i = 0; i < out.baz_len; i++) { printf("%s\n", out.baz[i]); } example_Foo_out_finish(&out);` Why does `vali` generate these twin structs, one for input parameters and the other for output parameters, instead of passing all parameters as function arguments? This does make calls slightly more verbose, but this has a few upsides: There is a clear split between input and output parameters, instead of having a variable number of function arguments for each. No need for the API user to remember where input parameters end and when output parameters begin, especially when there are a lot of these. On the wire and in the interface definition file, input and output parameters are objects. `vali` always generates structs for all objects. This is more consistent. If a new backwards-compatible version of the interface is published, the newly generated code is also backwards-compatible: old callers will still compile and work fine against the newly generated code. For instance, if a new optional field is added to the input parameters, it will naturally left as `NULL` by the caller when re-generating the code (because omitted fields are zero-initialized in C).

Service side with code generation The service side is more complicated because it needs to handle multiple connections concurrently and needs to be asynchronous. Being asynchronous is important to not block other clients when processing a call. The generator for the service code spits out one struct per method and a function to send a reply (and destroy the call): `struct example_Foo_service_call { struct vali_service_call *base; }; void example_Foo_close_with_reply(struct example_Foo_service_call call, const struct example_Foo_out *params);` The per-call struct wrapping the struct `vali_service_call *` makes functions sending replies strongly tied to a particular call, and provides type safety: a Foo reply cannot be sent to a Bar call. Additionally, the generator also provides a handler struct with one callback per method, and a function to obtain a generic handler from an interface handler: `struct example_handler { void (*Foo)(struct example_Foo_service_call call, const struct example_in *in); }; struct vali_service_call_handler example_get_call_handler(const struct example_handler *handler);` To use all of these toys, a service implementation can define a handler for the Foo method, then feed the result of `example_get_call_handler()` to `vali_service_set_call_handler()`: `static void handle_foo(struct example_Foo_service_call call, const struct example_Foo_in *in) { printf("Foo called with bar=%d\n", in->bar); example_Foo_close_with_reply(call, &(const struct example_Foo_out){ .baz = (char *[]){ "hello", "world" }, .baz_len = 2, }); } static const struct example_handler example_handler = { .Foo = handle_foo, }; int main(int argc, void *argv[]) { struct vali_service *service = vali_service_create(); vali_service_set_call_handler(service, example_get_call_handler(&example_handler));`

`vali_service_listen_unix(service, "/run/org.example.ftl"); while (vali_service_dispatch(service)); }` Service registry Some more elaborated services might want to implement more than a single interface. Additionally, services might want to add support for the `org.varlink.service` interface, which provides introspection: a client can query metadata about the service (e.g. service name, version) and the definition of each interface. `vali` makes this easy thanks to struct `vali_registry`. A service can initialize a new registry via `vali_registry_create()`, then register each interface by passing its definition and handler to `vali_registry_add()`. The generated code exposes the interface definition as an `example_interface` constant. Finally, the registry can be wired up to the struct `vali_service` by feeding the result of `vali_registry_get_call_handler()` to `vali_service_set_call_handler()`.

```
const struct vali_registry_options registry_options = { .vendor = "emersion", .product = "example", .version = "1.0.0", .url = "https://example.org", }; struct vali_registry *registry = vali_registry_create(&registry_options); vali_registry_add(registry, &example_interface, example_get_call_handler(&example_handler)); vali_registry_add(registry, &another_example_interface, another_example_get_call_handler(&another_example_handler)); struct vali_service *service = vali_service_create(); vali_service_set_call_handler(service, vali_registry_get_call_handler(registry)); vali_service_listen_unix(service, "/run/org.example.ftl"); while (vali_service_dispatch(service));
```

This is where the struct `vali_service_call_handler` fat pointer really shines: the wire-level struct `vali_service` and the higher-level registry can stay entirely separate. `struct vali_service` invokes the registry's handler, then the registry is responsible for routing the call to the correct interface-specific handler. The registry's internal state remains hidden away in the handler's opaque user data pointer. A complete client and service example is available in `vali's example/` directory. What's next? I plan to leverage `vali` in the next version of the `kanshi` Wayland output management daemon. We've discussed about `async` on the service side above, but we haven't discussed `async` on the client side. That can be useful too, especially when a client needs to juggle with multiple sockets, and is still a `TODO`. Something I'm still unhappy about is the lack of `const` fields generated structs. Let's have a look at the struct for output parameters given above: `struct example_Foo_out { char **baz; size_t baz_len; };` If a service has a bunch of `const char *` variables it wants to send as part of the reply, it needs to cast them to `char *` or `strdup()` them. None of these options are great. `static const char hiya[] = "hiya"; static void handle_foo(struct example_Foo_service_call call, const struct example_Foo_in *in) { example_Foo_close_with_reply(call, &(const struct example_Foo_out){ // Type error: implicit cast from "const char *" to "char *" .baz = (char *[]){ hiya }, .baz_len = 1, }); }` On the other hand, making all struct fields `const` would be cumbersome when dynamically constructing nested structs in replies, and would be a bit of a lie when passing a reply to `example_Foo_out_finish()` (that function frees all fields). Generating two structs (one `const`, one not) is not an option since types are shared between client and service, and some types can be referenced from both a call input and another call's output. Ideally, C would provide a way to propagate `const`-ness to fields, but that's not a thing. Oh well, that's life. If you need an IPC mechanism for your tool, please consider giving `vali` a shot! Feel free to reach out to report any bugs, questions or suggestions.

- [Iago Toral: XDC 2025](#) (2025/10/03 06:58)

It has been a while since my last post, I know. Today I just want to thank Igalia for continuing to give me and many other Igalians the opportunity to attend XDC. I had a great time in Vienna where I was able to catch up with other Mesa developers (including Igalians!) I rarely have the opportunity to see face to face. It is amazing to see how Mesa continues to gain traction and interest year after year, seeing more actors and vendors getting involved in one way or another... the push for open source drivers in the industry is real and it is fantastic to see it happening. I'd also like to thank the organization, I know all the work that goes into making these things happen, so big thanks to everyone who was involved, and to the speakers, the XDC program is getting better every year. Looking forward to next year already

- [Hans de Goede: Fedora 43 will ship with FOSS Meteor, Lunar and Arrow Lake MIPI camera support](#) (2025/09/30 18:55)

Good news the just released 6.17 kernel has support for the IPU7 CSI2 receiver and the missing USBIO drivers have recently landed in linux-next. I have backported the USBIO drivers + a few other camera fixes to the Fedora 6.17 kernel. I've also prepared an updated libcamera-0.5.2 Fedora package with support for IPU7 (Lunar Lake) CSI2 receivers as well as backporting a set of upstream SwStats and AGC fixes, fixing various crashes as well as the bad flicker MIPI camera users have been hitting with libcamera 0.5.2. Together these 2 updates should make Fedora 43's FOSS MIPI camera support work on most Meteor Lake, Lunar Lake and Arrow Lake laptops! If you want to give this a try, install / upgrade to Fedora 43 beta and install all updates. If you've installed rpmfusion's binary IPU6 stack please run: `sudo dnf remove akmod-intel-ipu6 'kmod-intel-ipu6*' to remove it as it may interfere with the FOSS stack and finally reboot. Please first try with qcama: sudo dnf install libcamera-qcamqcama which only tests libcamera and after that give apps which use the camera through pipewire a try like gnome's "Camera" app (snapshot) or video-conferencing in Firefox. Note snapshot on Lunar Lake triggers a bug in the LNL Vulkan code, to avoid this start snapshot from a terminal with: GSK_RENDERER=gl snapshot If you have a MIPI camera which still does not work please file a bug following these instructions and drop me an email with the bugzilla link at hansg@kernel.org. comments`

- [Sebastian Wick: XDG Intents Updates](#) (2025/09/24 16:57)

Andy Holmes wrote an excellent overview of XDG Intents in his "Best Intentions" blog post, covering the foundational concepts and early proposals. Unfortunately, due to GNOME Foundation issues, this work never fully materialized. As I have been running into more and more cases where this would provide a useful primitive for other features, I tried to continue the work. The specifications have evolved as I worked on implementing them in glib, desktop-file-utils and ptyxis. Here's what's changed: Intent-Apps Specification Andy showed this hypothetical syntax for scoped preferences: `[Default Applications] org.freedesktop.Thumbnailer=org.gimp.GIMP org.freedesktop.Thumbnailer[image/svg+xml]=org.gnome.Loupe;org.gimp.GIMP` We now use separate groups instead: `[Default Applications] org.freedesktop.Thumbnailer=org.gimp.GIMP [org.freedesktop.Thumbnailer] image/svg+xml=org.gnome.Loupe;org.gimp.GIMP` This approach creates a dedicated group for each intent, with keys representing the scopes. This way, we do not have to abuse the square brackets which were meant for translatable keys and allow only very limited values. The updated specification also adds support for intent.cache files to improve performance, containing up-to-date lists of applications supporting particular intents and scopes. This is very similar to the already existing cache for MIME types. The update-desktop-database tool is responsible for keeping the cache up-to-date. This is implemented in glib!4797, desktop-file-utils!27, and the updated specification is in xdg-specs!106. Terminal Intent Specification While Andy mentioned the terminal intent as a use case, Zander Brown tried to upstream the intent in xdg-specs!46 multiple years ago. However, because it depended on the intent-apps specification, it unfortunately never went anywhere. With the fleshed-out version of the intent-apps specification, and an implementation in glib, I was able to implement the terminal-intent specification in glib as well. With some help from Christian, we also added support for the intent in the ptyxis terminal. This revealed some shortcomings in the proposed D-Bus interface. In particular, when a desktop file gets activated with multiple URIs, and the Exec line in the desktop entry only indicates support for a limited number of URIs, multiple commands need to be launched. To support opening those commands in a single window but in multiple tabs in the terminal emulator, for example, those multiple commands must be part of a single D-Bus method call. The resulting D-Bus interface looks like this: `<interface name="org.freedesktop.Terminal1"> <method name="LaunchCommand"> <arg type='aa{sv}' name='commands' direction='in' /> <arg type='ay' name='desktop_entry' direction='in' /> <arg type='a{sv}' name='options' direction='in' /> <arg type='a{sv}' name='platform_data' direction='in' /> </method> </interface>` This is

implemented in glib!4797, ptyxis!119 and the updated specification is in xdg-specs!107. Deeplink Intent Andy's post discussed a generic "org.freedesktop.UriHandler" with this example: [org.freedesktop.UriHandler] Supports=wise.com; Patterns=https://*.wise.com/link?urn=urn%3Awise%3Atransfers; The updated specification introduces a specific org.freedesktop.handler.Deeplink1 intent where the scheme is implicitly http or https and the host comes from the scope (i.e., the Supports part). The pattern matching is done on the path alone: [org.freedesktop.handler.Deeplink1] Supports=example.org;extensions.gnome.org example.org=/login;/test/a?a extensions.gnome.org=/extension/*/*/install;/extension/*/*/uninstall This allows us to focus on deeplinking alone and allows the user to set the order of handlers for specific hosts. In this example, the app would handle the URIs http://example.org/login, http://example.org/test/aba, http://extensions.gnome.org/extension/123456/BestExtension/install and so on. There is a draft implementation in glib!4833 and the specification is in xdg-specs!109. Deeplinking Issues and Other Handlers I am still unsure about the Deeplink1 intent. Does it make sense to allow schemes other than http and https? If yes, how should the priority of applications be determined when opening a URI? How complex does the pattern matching need to be? Similarly, should we add an org.freedesktop.handler.Scheme1 intent? We currently abuse MIME handlers for this, so it seems like a good idea, but then we need to take backwards compatibility into account. Maybe we can modify update-desktop-database to add entries from org.freedesktop.handler.Scheme1 to mimeapps.list for that? If we go down that route, is there a reason not to also do the same for MIME handlers and add an org.freedesktop.handler.Mime1 intent for that purpose with the same backwards compatibility mechanism? Deeplinking to App Locations While working on this, I noticed that we are not great at allowing linking to locations in our apps. For example, most email clients do not have a way to link to a specific email. Most calendars do not allow referencing a specific event. Some apps do support this. For example, Zotero allows linking to items in the app with URIs of the form zotero://select/items/0_USN95MJC. Maybe we can improve on this? If all our apps used a consistent scheme and queries (for example xdg-app-org.example.appid:/some/path/in/the/app?name=Example), we could render those links differently and finally have a nice way to link to an email in our calendar. This definitely needs more thought, but I do like the idea. Security Considerations Allowing apps to describe more thoroughly which URIs they can handle is great, but we also live in a world where security has to be taken into account. If an app wants to handle the URI https://bank.example.org, we better be sure that this app actually is the correct banking app. This unfortunately is not a trivial issue, so I will leave it for the next time.

- [Sebastian Wick: Integrating libdex with GDBus](#) (2025/09/18 18:58)

Writing asynchronous code in C has always been a challenge. Traditional callback-based approaches, including GLib's async/finish pattern, often lead to the so-called callback hell that's difficult to read and maintain. The libdex library offers a solution to this problem, and I recently worked on expanding the integration with GLib's GDBus subsystem. The Problem with the Sync and Async Patterns Writing C code involving tasks which can take non-trivial amount of time has traditionally required choosing between two approaches: Synchronous calls - Simple to write but block the current thread Asynchronous callbacks - Non-blocking but result in callback hell and complex error handling Often the synchronous variant is chosen to keep the code simple, but in a lot of cases, blocking for potentially multiple seconds is not acceptable. Threads can be used to prevent the other threads from blocking, but it creates parallelism and with it the need for locking. It also can potentially create a huge amount of threads which mostly sit idle. The asynchronous variant has none of those problems, but consider a typical async D-Bus operation in traditional GLib code: static void on_ping_ready (GObject *source_object, GAsyncResult *res, gpointer data) { g_autofree char *pong = NULL; if (!dex_dbus_ping_pong_call_ping_finish (DEX_BUS_PING_PONG (source_object), &pong, res, NULL)) return; // handle error g_print ("client: %s\n",

```

pong); } static void on_ping_pong_proxy_ready (GObject *source_object, GAsyncResult *res, gpointer data) { DexDBusPingPong *pp
dex_dbus_ping_pong_proxy_new_finish (res, NULL); if (!pp) return; // Handle error dex_dbus_ping_pong_call_ping (pp, "ping", NULL,
on_ping_ready, NULL); } This pattern becomes unwieldy quickly, especially with multiple operations, error handling, shared data and cleanup
across multiple callbacks. What is libdex? Dex provides Future-based programming for GLib. It provides features for application and library
authors who want to structure concurrent code in an easy to manage way. Dex also provides Fibers which allow writing synchronous looking code
in C while maintaining the benefits of asynchronous execution. At its core, libdex introduces two key concepts: Futures: Represent values that
will be available at some point in the future Fibers: Lightweight cooperative threads that allow writing synchronous-looking code that yields
control when waiting for asynchronous operations Futures alone already simplify dealing with asynchronous code by specefyng a call chain
(dex_future_then(), dex_future_catch(), and dex_future_finally()), or even more elaborate flows (dex_future_all(), dex_future_all_race(),
dex_future_any(), and dex_future_first()) at one place, without the typical callback hell. It still requires splitting things into a bunch of functions
and potentially moving data through them. static DexFuture * lookup_user_data_cb (DexFuture *future, gpointer user_data) { g_autoptr(MyUser)
user = NULL; g_autoptr(GError) error = NULL; // the future in this cb is already resolved, so this just gets the value // no fibers involved user =
dex_await_object (future, &error); if (!user) return dex_future_new_for_error (g_steal_pointer (&error)); return dex_future_first
(dex_timeout_new_seconds (60), dex_future_any (query_db_server (user), query_cache_server (user), NULL), NULL); } static void print_user_data
(void) { g_autoptr(DexFuture) future = NULL; future = dex_future_then (find_user (), lookup_user_data_cb, NULL, NULL); future = dex_future_then
(future, print_user_data_cb, NULL, NULL); future = dex_future_finally (future, quit_cb, NULL, NULL); g_main_loop_run (main_loop); } The real
magic of libdex however lies in fibers and the dex_await() function, which allows you to write code that looks synchronous but executes
asynchronously. When you await a future, the current fiber yields control, allowing other work to proceed while waiting for the result.
g_autoptr(MyUser) user = NULL; g_autoptr(MyUserData) data = NULL; g_autoptr(GError) error = NULL; user = dex_await_object (find_user (),
&error); if (!user) return dex_future_new_for_error (g_steal_pointer (&error)); data = dex_await_boxed (dex_future_first
(dex_timeout_new_seconds (60), dex_future_any (query_db_server (user), query_cache_server (user), NULL), NULL), &error); if (!data) return
dex_future_new_for_error (g_steal_pointer (&error)); g_print ("%s", data->name); Christian Hergert wrote pretty decent documentation, so check
it out! Bridging libdex and GDBus With the new integration, you can write D-Bus client code that looks like this: g_autoptr(DexDBusPingPong) *pp
= NULL; g_autoptr(DexDBusPingPongPingResult) result = NULL; pp = dex_await_object (dex_dbus_ping_pong_proxy_new_future (connection,
G_DBUS_PROXY_FLAGS_NONE, "org.example.PingPong", "/org/example/pingpong"), &error); if (!pp) return dex_future_new_for_error
(g_steal_pointer (&error)); res = dex_await_boxed (dex_dbus_ping_pong_call_ping_future (pp, "ping"), &error); if (!res) return
dex_future_new_for_error (g_steal_pointer (&error)); g_print ("client: %s\n", res->pong); This code is executing asynchronously, but reads like
synchronous code. Error handling is straightforward, and there are no callbacks involved. On the service side, if enabled, method handlers will
run in a fiber and can use dex_await() directly, enabling complex asynchronous operations within service implementations: static gboolean
handle_ping (DexDBusPingPong *object, GDBusMethodInvocation *invocation, const char *ping) { g_print ("service: %s\n", ping); dex_await
(dex_timeout_new_seconds (1), NULL); dex_dbus_ping_pong_complete_ping (object, invocation, "pong"); return
G_DBUS_METHOD_INVOCATION_HANDLED; } static void dex_dbus_ping_pong_iface_init (DexDBusPingPongIface *iface) { iface->handle_ping =
handle_ping; } pp = g_object_new (DEX_TYPE_PING_PONG, NULL); dex_dbus_interface_skeleton_set_flags (DEX_DBUS_INTERFACE_SKELETON
(pp), DEX_DBUS_INTERFACE_SKELETON_FLAGS_HANDLE_METHOD_INVOCATIONS_IN_FIBER); This method handler includes a 1-second delay, but

```

instead of blocking the entire service, it yields control to other fibers during the timeout. The merge request contains a complete example of a client and service communicating with each other. Implementation Details The integration required extending GDBus's code generation system. Rather than modifying it directly, the current solution introduces a very simple extension system to GDBus' code generation. The generated code includes: Future-returning functions: For every `_proxy_new()` and `_call_$method()` function, corresponding `_future()` variants are generated Result types: Method calls return boxed types containing all output parameters Custom skeleton base class: Generated skeleton classes inherit from `DexDBusInterfaceSkeleton` instead of `GDBusInterfaceSkeleton`, which implements dispatching method handlers in fibers Besides the GDBus code generation extension system, there are a few more changes required in GLib to make this work. This is not merged at the time of writing, but I'm confident that we can move this forward. Future Directions I hope that this work convinces more people to use libdex! We have a whole bunch of existing code bases which will have to stick with C in the foreseeable future, and libdex provides tools to make incremental improvements. Personally, I want to start using in in the `xdg-desktop-portal` project.

- [Mike Blumenkrantz: Now We CAD](#) (2025/09/16 00:00)

Perf Must Increase. After my last post, I'm sure everyone was speculating about the forthcoming zink takeover of the CAD industry. Or maybe just wondering why I'm bothering with this at all. Well, the answer is simple: CAD performance is all performance. If I can improve FPS in viewperf, I'm decreasing CPU utilization in all apps, which is generally useful. As in the previous post, the `catia` section of viewperf was improved to a whopping 34fps against the reference driver (`radeonsi`) by eliminating a few hundred thousand atomic operations per frame. An interesting observation here is that while eliminating atomic operations in `radeonsi` does improve FPS there by ~5% (105fps), there is no bottlenecking, so this does not "unlock" further optimizations in the same way that it does for zink. I speculate this is because zink has `radv` underneath, which affects memory access across `ccx` in ways that do not affect `radeonsi`. In short: a rising tide lifts all ships in the harbor, but since zink was effectively a sunken ship, it is rising much more than the others. Even More Improvements Since that previous post, I and others have been working quietly in the background on other improvements, all of which have landed in mesa main already: A nice 35% improvement, largely from three MRs: zink draw optimizations zink vbo binding optimizations `radv` dynamic state optimizations That's right. In my quest to maximize perf, I have roped in veteran `radv` developer and part-time vacation enthusiast, Samuel Pitoiset. Because `radv` is slow. `vkoverhead` exists to target noticeably slow cases, and by harnessing the forbidden power of rewriting the whole driver, it was possible for a lone Frenchman to significantly reduce bottlenecking during draw emission. This Isn't Even My Final Form Obviously. I'm not about to say that I'll only stop when I reach performance parity, but the FPS can still go up. At this point, however, it's becoming less useful (in zink) to look at flamegraphs. There's only so much optimization that can be done once the code has been simplified to a certain extent, and eventually those optimizations will lead to obfuscated code which is harder to maintain. Thus, it's time to step back and look architecturally. What is the app doing? How does that reach the driver? Can it be improved? `GALLIUM_TRACE` is a great tool for this, as it logs the API stream as it reaches the backend driver, and there are parser tools to convert the output XML to something readable. Let's take a look at a small cross-section of the trace: `pipe_context::set_vertex_buffers(pipe = context_2, num_buffers = 2, buffers = [[is_user_buffer = 0, buffer_offset = 0, buffer.resource = resource_10043], [is_user_buffer = 0, buffer_offset = 7440, buffer.resource = resource_10043]])` `pipe_context::draw_vbo(pipe = context_2, info = [index_size = 2, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 0, max_index = 1257, primitive_restart = 0, restart_index = 0, index.resource = resource_10044], drawid_offset = 0, indirect = NULL, draws = [[start = 0, count = 1257, index_bias = 0]], num_draws = 1)` `pipe_context::set_vertex_buffers(pipe = context_2, num_buffers = 2, buffers = [[is_user_buffer = 0, buffer_offset = 0, buffer.resource =`

```
resource_10045], [is_user_buffer = 0, buffer_offset = 7632, buffer.resource = resource_10045])) pipe_context::draw_vbo(pipe = context_2, info =
[index_size = 2, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 0, max_index = 1257, primitive_restart =
0, restart_index = 0, index.resource = resource_10046], drawid_offset = 0, indirect = NULL, draws = [[start = 0, count = 1257, index_bias = 0]],
num_draws = 1) pipe_context::set_vertex_buffers(pipe = context_2, num_buffers = 2, buffers = [[is_user_buffer = 0, buffer_offset = 0,
buffer.resource = resource_10047], [is_user_buffer = 0, buffer_offset = 7680, buffer.resource = resource_10047]]) pipe_context::draw_vbo(pipe
= context_2, info = [index_size = 2, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 0, max_index = 1257,
primitive_restart = 0, restart_index = 0, index.resource = resource_10048], drawid_offset = 0, indirect = NULL, draws = [[start = 0, count =
1257, index_bias = 0]], num_draws = 1) pipe_context::set_vertex_buffers(pipe = context_2, num_buffers = 2, buffers = [[is_user_buffer = 0,
buffer_offset = 0, buffer.resource = resource_10049], [is_user_buffer = 0, buffer_offset = 7656, buffer.resource = resource_10049]])
pipe_context::draw_vbo(pipe = context_2, info = [index_size = 2, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1,
min_index = 0, max_index = 1257, primitive_restart = 0, restart_index = 0, index.resource = resource_10050], drawid_offset = 0, indirect =
NULL, draws = [[start = 0, count = 1257, index_bias = 0]], num_draws = 1) pipe_context::set_vertex_buffers(pipe = context_2, num_buffers = 2,
buffers = [[is_user_buffer = 0, buffer_offset = 0, buffer.resource = resource_10051], [is_user_buffer = 0, buffer_offset = 7752, buffer.resource =
resource_10051]]) pipe_context::draw_vbo(pipe = context_2, info = [index_size = 2, has_user_indices = 0, mode = 5, start_instance = 0,
instance_count = 1, min_index = 0, max_index = 1257, primitive_restart = 0, restart_index = 0, index.resource = resource_10052], drawid_offset
= 0, indirect = NULL, draws = [[start = 0, count = 1257, index_bias = 0]], num_draws = 1) pipe_context::set_vertex_buffers(pipe = context_2,
num_buffers = 2, buffers = [[is_user_buffer = 0, buffer_offset = 0, buffer.resource = resource_10053], [is_user_buffer = 0, buffer_offset = 7800,
buffer.resource = resource_10053]]) pipe_context::draw_vbo(pipe = context_2, info = [index_size = 2, has_user_indices = 0, mode = 5,
start_instance = 0, instance_count = 1, min_index = 0, max_index = 1257, primitive_restart = 0, restart_index = 0, index.resource =
resource_10054], drawid_offset = 0, indirect = NULL, draws = [[start = 0, count = 1257, index_bias = 0]], num_draws = 1)
pipe_context::set_vertex_buffers(pipe = context_2, num_buffers = 2, buffers = [[is_user_buffer = 0, buffer_offset = 0, buffer.resource =
resource_10055], [is_user_buffer = 0, buffer_offset = 7968, buffer.resource = resource_10055]]) pipe_context::draw_vbo(pipe = context_2, info =
[index_size = 2, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 0, max_index = 1257, primitive_restart =
0, restart_index = 0, index.resource = resource_10056], drawid_offset = 0, indirect = NULL, draws = [[start = 0, count = 1257, index_bias = 0]],
num_draws = 1) pipe_context::set_vertex_buffers(pipe = context_2, num_buffers = 2, buffers = [[is_user_buffer = 0, buffer_offset = 0,
buffer.resource = resource_10057], [is_user_buffer = 0, buffer_offset = 7968, buffer.resource = resource_10057]]) pipe_context::draw_vbo(pipe
= context_2, info = [index_size = 2, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 0, max_index = 1257,
primitive_restart = 0, restart_index = 0, index.resource = resource_10058], drawid_offset = 0, indirect = NULL, draws = [[start = 0, count =
1257, index_bias = 0]], num_draws = 1) pipe_context::set_vertex_buffers(pipe = context_2, num_buffers = 2, buffers = [[is_user_buffer = 0,
buffer_offset = 0, buffer.resource = resource_10059], [is_user_buffer = 0, buffer_offset = 8136, buffer.resource = resource_10059]])
pipe_context::draw_vbo(pipe = context_2, info = [index_size = 2, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1,
min_index = 0, max_index = 1257, primitive_restart = 0, restart_index = 0, index.resource = resource_10060], drawid_offset = 0, indirect =
NULL, draws = [[start = 0, count = 1257, index_bias = 0]], num_draws = 1) pipe_context::set_vertex_buffers(pipe = context_2, num_buffers = 2,
buffers = [[is_user_buffer = 0, buffer_offset = 0, buffer.resource = resource_10061], [is_user_buffer = 0, buffer_offset = 8280, buffer.resource =
```

```
resource_10061])) pipe_context::draw_vbo(pipe = context_2, info = [index_size = 2, has_user_indices = 0, mode = 5, start_instance = 0,
instance_count = 1, min_index = 0, max_index = 1257, primitive_restart = 0, restart_index = 0, index.resource = resource_10062], drawid_offset
= 0, indirect = NULL, draws = [[start = 0, count = 1257, index_bias = 0]], num_draws = 1) pipe_context::set_vertex_buffers(pipe = context_2,
num_buffers = 2, buffers = [[is_user_buffer = 0, buffer_offset = 0, buffer.resource = resource_10063], [is_user_buffer = 0, buffer_offset = 8040,
buffer.resource = resource_10063]]) pipe_context::draw_vbo(pipe = context_2, info = [index_size = 2, has_user_indices = 0, mode = 5,
start_instance = 0, instance_count = 1, min_index = 0, max_index = 1257, primitive_restart = 0, restart_index = 0, index.resource =
resource_10064], drawid_offset = 0, indirect = NULL, draws = [[start = 0, count = 1257, index_bias = 0]], num_draws = 1)
pipe_context::set_vertex_buffers(pipe = context_2, num_buffers = 2, buffers = [[is_user_buffer = 0, buffer_offset = 0, buffer.resource =
resource_10065], [is_user_buffer = 0, buffer_offset = 7608, buffer.resource = resource_10065]]) pipe_context::draw_vbo(pipe = context_2, info =
[index_size = 2, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 0, max_index = 1257, primitive_restart =
0, restart_index = 0, index.resource = resource_10066], drawid_offset = 0, indirect = NULL, draws = [[start = 0, count = 1257, index_bias = 0]],
num_draws = 1) As expected, a huge chunk of the runtime is just set_vertex_buffers -> draw_vbo. Architecturally, this leads to a lot of
unavoidably wasted cycles in drivers: set_vertex_buffers “binds” vertex buffers to the context and flags state updates draw_vbo checks all of the
driver’s update-able states, updates the flagged ones, and then emits draws But in the scenario where the driver can know ahead of time exactly
what states will be updated, couldn’t that yield an improvement? For example, bundling these two calls into a single draw call would eliminate:
“binding” of vertex buffers vbo state update flagging draw-time validation calling multiple driver entrypoints In theory, it seems like this should
be pretty good. And now that vertex buffer lifetimes have been reworked to use explicit ownership rather than garbage collection, it’s actually
possible to do this. The optimal site for the optimization would be in threaded-context, where similar types of draw merging are already
occurring. The result looks something like this in a comparable trace: pipe_context::draw_vbo_buffers(pipe = pipe_2, info = [index_size = 0,
has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 1141, max_index = 5, primitive_restart = 0, restart_index
= 0, index.resource = NULL], buffer_count = 3, buffers = [[is_user_buffer = 0, buffer_offset = 163536, buffer.resource = resource_30210],
[is_user_buffer = 0, buffer_offset = 191032, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 771328, buffer.resource =
resource_29602]], draws = [[start = 1141, count = 5]], num_draws = 1) pipe_context::draw_vbo_buffers(pipe = pipe_2, info = [index_size = 0,
has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 1146, max_index = 5, primitive_restart = 0, restart_index
= 0, index.resource = NULL], buffer_count = 3, buffers = [[is_user_buffer = 0, buffer_offset = 218528, buffer.resource = resource_30210],
[is_user_buffer = 0, buffer_offset = 246144, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 771360, buffer.resource =
resource_29602]], draws = [[start = 1146, count = 5]], num_draws = 1) pipe_context::draw_vbo_buffers(pipe = pipe_2, info = [index_size = 0,
has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 1151, max_index = 5, primitive_restart = 0, restart_index
= 0, index.resource = NULL], buffer_count = 3, buffers = [[is_user_buffer = 0, buffer_offset = 273760, buffer.resource = resource_30210],
[is_user_buffer = 0, buffer_offset = 301496, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 771392, buffer.resource =
resource_29602]], draws = [[start = 1151, count = 5]], num_draws = 1) pipe_context::draw_vbo_buffers(pipe = pipe_2, info = [index_size = 0,
has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 1156, max_index = 5, primitive_restart = 0, restart_index
= 0, index.resource = NULL], buffer_count = 3, buffers = [[is_user_buffer = 0, buffer_offset = 329232, buffer.resource = resource_30210],
[is_user_buffer = 0, buffer_offset = 357088, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 771424, buffer.resource =
```

resource_29602]], draws = [[start = 1156, count = 5]], num_draws = 1) pipe_context::draw_vbo_buffers(pipe = pipe_2, info = [index_size = 0, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 1161, max_index = 5, primitive_restart = 0, restart_index = 0, index.resource = NULL], buffer_count = 3, buffers = [[is_user_buffer = 0, buffer_offset = 384944, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 412920, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 771456, buffer.resource = resource_29602]], draws = [[start = 1161, count = 5]], num_draws = 1) pipe_context::draw_vbo_buffers(pipe = pipe_2, info = [index_size = 0, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 1166, max_index = 5, primitive_restart = 0, restart_index = 0, index.resource = NULL], buffer_count = 3, buffers = [[is_user_buffer = 0, buffer_offset = 440896, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 468992, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 771488, buffer.resource = resource_29602]], draws = [[start = 1166, count = 5]], num_draws = 1) pipe_context::draw_vbo_buffers(pipe = pipe_2, info = [index_size = 0, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 1171, max_index = 5, primitive_restart = 0, restart_index = 0, index.resource = NULL], buffer_count = 3, buffers = [[is_user_buffer = 0, buffer_offset = 497088, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 525304, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 771520, buffer.resource = resource_29602]], draws = [[start = 1171, count = 5]], num_draws = 1) pipe_context::draw_vbo_buffers(pipe = pipe_2, info = [index_size = 0, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 1176, max_index = 11, primitive_restart = 0, restart_index = 0, index.resource = NULL], buffer_count = 3, buffers = [[is_user_buffer = 0, buffer_offset = 553520, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 582000, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 771552, buffer.resource = resource_29602]], draws = [[start = 1176, count = 11]], num_draws = 1) pipe_context::draw_vbo_buffers(pipe = pipe_2, info = [index_size = 0, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 1187, max_index = 5, primitive_restart = 0, restart_index = 0, index.resource = NULL], buffer_count = 3, buffers = [[is_user_buffer = 0, buffer_offset = 610480, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 639080, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 771584, buffer.resource = resource_29602]], draws = [[start = 1187, count = 5]], num_draws = 1) pipe_context::draw_vbo_buffers(pipe = pipe_2, info = [index_size = 0, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 1192, max_index = 6, primitive_restart = 0, restart_index = 0, index.resource = NULL], buffer_count = 3, buffers = [[is_user_buffer = 0, buffer_offset = 667680, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 696424, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 771616, buffer.resource = resource_29602]], draws = [[start = 1192, count = 6]], num_draws = 1) pipe_context::draw_vbo_buffers(pipe = pipe_2, info = [index_size = 0, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 1198, max_index = 5, primitive_restart = 0, restart_index = 0, index.resource = NULL], buffer_count = 3, buffers = [[is_user_buffer = 0, buffer_offset = 725168, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 754032, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 771648, buffer.resource = resource_29602]], draws = [[start = 1198, count = 5]], num_draws = 1) pipe_context::draw_vbo_buffers(pipe = pipe_2, info = [index_size = 0, has_user_indices = 0, mode = 5, start_instance = 0, instance_count = 1, min_index = 1203, max_index = 5, primitive_restart = 0, restart_index = 0, index.resource = NULL], buffer_count = 3, buffers = [[is_user_buffer = 0, buffer_offset = 782896, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 811880, buffer.resource = resource_30210], [is_user_buffer = 0, buffer_offset = 771680, buffer.resource = resource_29602]], draws = [[start = 1203, count = 5]], num_draws = 1) It's more compact, which is nice, but how does the perf look? About another 40% improvement, now over 60fps: nearly double the endpoint of the last post. Huge. And this is driving ecosystem improvements which

will affect other apps and games which don't even use zink. Stay winning, Open Source graphics.

- [Dave Airlie \(blogspot\): radv takes over from AMDVLK](#) (2025/09/15 19:08)

AMD have announced the end of the AMDVLK open driver in favour of focusing on radv for Linux use cases. When Bas and I started radv in 2016, AMD were promising their own Linux vulkan driver, which arrived in Dec 2017. At this point radv was already shipping in most Linux distros. AMD strategy of having AMDVLK was developed via over the wall open source releases from internal closed development was always going to be a second place option at that point. When Valve came on board and brought dedicated developer power to radv, and the aco compiler matured, there really was no point putting effort into using AMDVLK which was hard to package and impossible to contribute to meaningfully for external developers. radv is probably my proudest contribution to the Linux ecosystem, finally disproving years of idiots saying an open source driver could never compete with a vendor provided driver, now it is the vendor provided driver. I think we will miss the open source PAL repo as a reference source and I hope AMD engineers can bridge that gap, but it's often hard to find workarounds you don't know exist to ask about them. I'm also hoping AMD will add more staffing beyond the current levels especially around hardware enablement and workarounds. Now onwards to NVK victory :-)[1] <https://github.com/GPUOpen-Drivers/AMDVLK/discussions/416>

- [Christian Schaller: More adventures in the land of AI and Open Source](#) (2025/09/09 14:39)

I been doing a lot of work with AI recently, both as part of a couple of projects I am part of at work, but I have also taken a personal interest in understanding the current state and what is possible. My favourite AI tool currently is Claude.ai. Anyway I have a Prusa Core One 3D printer now that I also love playing with and one thing I been wanting to do is to print some multicolor prints with it. So the Prusa Core One is a single extruder printer, which means it only has 1 filament loaded at any given time. Other printers on the market, like the PrusaXL has 5 extruders, so it can have 5 filaments or colors loaded at the same time. Prusa Single Extruder Multimaterial setting The thing is that the Prusa Slicer (the slicer is the software that takes a 3d model and prepares the instructions for the printer based on that 3d model) got this feature called Single Extruder Multi Material. And while it is a process that wastes a lot of filament and takes a lot of manual intervention during the print, it does basically work. What I quickly discovered was that using this feature is non-trivial. First of all I had to manually add some G Code to the model to actually get it to ask me to switch filament for each color in my print, but the bigger issue is that the printer will ask you to change the color or filament, but you have no way of knowing which one to switch to, so for my model I had 15 filament changes and no simple way of knowing which order to switch in. So people where solving this among other things through looking through the print layer by layer and writing down the color changes, but I thought that this must be possible to automate with an application. So I opened Claude and started working on this thing I ended up calling Prusa Color Mate.. So the idea for the application was simple enough, have it analyze the project file, extract information about the order of color changes and display them for the user in a way that allows them to manually check of each color as its inserted. So I started off with doing a simple python script that would just print to the console. So it quickly turned out that the hard part of this project was to parse the input files and it was made worse by my ignorance. So what I learned the hard way is that if you store a project in Prusa Slicer it will use this format called 3mf. So my thought was, lets just analyze the 3mf file and extract the information I need. It took my quite a bit of back and forth with Claude, feeding claude source code from Prusa's implementation and pdf files with specifications, but eventually the application did spit out a list of 15 toolchanges and the colors associated with them. So I happily tried to use it to print my model. I quickly discovered that the color ordering was all wrong. And after even more back and forth with Claude and reading online I realized that the 3mf file is a format for storing 3d models, but that is not what is being fed your 3d printer, instead for the printer the file provided is a bgcode file. And while the 3mf file did contain the information

that you had to change filament 15 times, the information on in which order is simply not stored in the 3mf file as that is something chosen as part of composing your print. That print composition file is using a file format called bgcode. So I now had to extract the information from the bgcode file which took me basically a full day to figure out with the help of Claude. I could probably have gotten over the finish line sooner by making some better choices underway, but the extreme optimism of the AI probably lead me to believe it was going to be easier than it was to for instance just do everything in Python. At first I tried using this libbgcode library written in C++, but I had a lot of issues getting Claude to incorporate it properly into my project, with Meson and CMAKE interaction issues (in retrospect I should have just made a quick RPM of libbgcode and used that). After a lot of struggles with this Claude thought that parsing the bgcode file in python natively would be easier than trying to use the C++ library, so I went down that route. I started by feeding Claude a description of the format that I found online and asked it to write me a parser for it. It didn't work very well and I ended up having a lot of back and forth, testing and debugging, finding more documentation, including a blog post about this meatpack format used inside the file, but it still didn't really work very well. In the end what probably helped the most was asking it to use the relevant files from libbgcode and Prusa Slicer as documentation, because even if that too took a lot of back and forth, eventually I had a working application that was able to extract the tool change data and associated colors from the file. I ended up using one external dependency which was the heatshrink2 library that I PIP installed, but while that worked correctly, it took a look time for me and Claude to figure out exactly what parameters to feed it to work with the Prusa generated file. Screenshot of Prusa Color Mate So now I had the working application going and was able to verify it with my first print. I even polished it up a little, by also adding detection of the manual filament change code, so that people who try to use the application will be made aware they need to add that through Prusa Slicer. Maybe I could bake that into the tool, but atm I got only bgcode decoders, not encoders, in my project. Warning showed for missing G Code Dialog that gives detailed instructions for how to add G Code So to conclude, it probably took me 2.5 days to write this application using Claude, it is a fairly niche tool, so I don't expect a lot of users, but I made it to solve a problem for myself. If I had to write this pre-AI myself it would have taken me weeks, like figuring out the different formats and how library APIs worked etc. would have taken me a long time. So I am not an especially proficient coder, so a better coder than me could probably put this together quicker than I would, but I think this is part of what I think will change with AI, that even with limited time and technical skills you can put together simple applications like this to solve your own problems. If you are a Prusa Core One user and would like to play with multicolor prints you can find Prusa Color Mate on Gitlab. I have not tested it on any other system or printer than my own, so I don't even know if it will work with other non-Core One Prusa printers. There are rpms for Fedora you can download in the packaging directory of the gitlab repo, which also includes a RPM for the heatshrink2 library. As for future plans for this application I don't really have any. It solves my issue the way it is today, but if there turns out to be an interested user community out there maybe I will try to clean it up and create a proper flatpak for it.

- [Mike Blumenkrantz: Big Lifts](#) (2025/09/09 00:00)

New Record For months now I've been writing increasingly unhinged patchsets. Sometimes it might even seem like there is no real point to what I'm doing. Or that I'm just churning code to have something to do. But I'm here today to tell you that finally, the long journey is over. We have reached the promised land of perf. Huge. Many months ago, I began examining viewperf, AKA the final frontier of driver performance. What makes this the final frontier? some of you might be asking. Imagine an application which does 10,000 individual draws per frame, each with their own vertex buffer bindings. That's a lot of draws. Now imagine an application which does ten times that many draws per frame. This is viewperf, which represents common use cases of CAD-adjacent technologies. Where other applications might hammer on the GPU, viewperf tests the CPU

utilization. It's what separates the real developers from average, sane people. So all those months ago, I ran viewperf on zink, and I ended up here: 18fps. This is on threadripper 5975WX with RADV; not the most modern or powerful CPU, but it's still pretty quick. Then I loaded up radeonsi and got 100fps. Brutal. Plumbing The Abyss Examining this was where I entered into into realms of insanity not known to mere mortals. perf started to fail and give confusing results, other profilers just drew a circle around the driver and pointed to the whole thing as the problem area, and some tools just gave up entirely. No changes affected the performance in any way. This is when the savvy hacker begins profiling by elimination: delete as much code as possible and try to force changes. Thus, I deleted a lot of code to see what would pop out, and eventually I discovered the horrifying truth: I was being bottlenecked by the sheer number of atomic operations occurring. Like I said before, viewperf does upwards of 100,000 draw calls per frame. This means 100,000 draw calls, 100,000 vertex buffer binds (times two because there are two vertex buffers), 100,000 index buffer binds, and a few shader changes sprinkled in. The way that mesa/gallium work means that every single vertex buffer and index buffer which get sent to the driver incur multiple atomic operations (each) along the way for refcounting: because gallium uses refcounting rather than an ownership model since it is much easier to manage. That means we're talking about upwards of 300,000 atomic operations per frame. Unfortunately, hackily deleting all the refcounting made the FPS go brrrrr, and it was a long road to legitimately get there. A very, very long road. Six months, in fact. But all the unhinged MRs above landed, reducing the surface area of the refcounting to just buffers, which put me in a position to do this pro gamer move where I also am removing all the refcounting from the buffers. This works, roughly speaking, by enforcing ownership on the buffers and then releasing them when they are no longer used. Sounds simple, but plumbing it through all the gallium drivers without breaking everything was less so. Let's see where moving to that model gets the numbers: One more frame. Tremendous. But wait, there's more. The other part of that MR further deletes all the refcounting in zink for buffers, fully removing the atomics. And... Blammo, that doubles the perf and manages to eliminate the bottleneck, which sets the stage for further improvements. The gap is still large, but it's about to close real fast. Shout out to Marek for heroically undertaking the review of this leviathan.

- [Mike Blumenkrantz: Mesh Shader Progress](#) (2025/09/05 00:00)

VKCTS Tests: 27,890 | GLCTS Tests: 227 | Percentage of Vulkan Drivers With Mesh Bugs: 100%

- [Hans de Goede: Leaving Red Hat](#) (2025/09/03 18:46)

After 17 years I feel that it is time to change things up a bit and for a new challenge. I'm leaving Red Hat and my last day at Red Hat will be October 31st. I would like to thank Red Hat for the opportunity to work on many interesting open-source projects during my time at Red Hat and for all the things I've learned while at Red Hat. I want to use this opportunity to thank everyone I've worked with, both my great Red Hat colleagues, as well as everyone from the community for all the good times during the last 17 years. I've a pretty good idea of what will come next, but this is not set in stone yet. I definitely will continue to work on open-source and on Linux hw-enablement. comments

- [Mike Blumenkrantz: Tiler Improvements](#) (2025/08/29 00:00)

Super Late Code Meant to blog about this last quarter, but somehow another two months went by and here we are. A while back, I did some work to improve zink performance on tiling GPUs. Namely this entailed adding renderpass tracking into threaded-context, and also implementing command stream reordering, and inlining swapchain resolves, and framebuffer discards, and actually maybe it's more than just "some" work. All of this amounted to improved performance by reducing memory bandwidth. How much improved performance? All of it. And then, around two months ago, a colleague told me he was no longer going to use zink on his tiling GPU. Devastated Some of you noticed that the blog has gone quiet in recent times. I'm going to take this opportunity to foist all the blame onto that colleague: to preserve his identity, let's just call him Gabe.

Gabe came to me a few months ago and told me zink was too slow. Vulkan was better. Faster. More “reliable”. I said there’s no way that could be true; I’ve put way more bugs into Vulkan than I have into zink. Unblinking, he stared at me across the digital divide. I task-switched to important whitespace cleanups. Time passed, and I pulled myself together. I compiled some app traces. Analyzed them. Did some deep thinking. There was one place where zink indeed could be less performant than this “Vulkan” thing. The final frontier of driver performance. Some call it graphics heaven. I call it hell. Web Browsers Chrome is the web browser, and, statistically, everyone uses it. It ships on desktops and phones, embeds in apps, and even allows you to read this blog. Haters will say No I uSe FiReFoX, but they may as well be Netscape users in the year 2000. In the past, Chrome defaulted to using GL, which made testing easy. Now, however, `--disable-features=Vulkan` is needed to return to the comfort of an API so reliable it no longer receives versioned updates. Looking at an apitrace of Chrome, I saw a disturbing rendering pattern that went something like this: draw some element on a page using multisampled FBO1 resolve FBO1 to texture1 composite texture1 onto larger FBO2/texture2 composite texture2 onto even larger, multisampled FBO3 resolve FBO3 to swapchain present In this case, zink would correctly inline the FBO3/swapchain resolve at the end, but the intermediate multisampled rendering on FBO1 would pay the full performance penalty of storing the multisampled image data and then loading it again for the separate resolve operation. I’d like to say it was simple to inline this intermediate resolve. That I just slapped a single MR into mesa and it magically worked. Unfortunately, nothing is ever that simple. There were minor fixups all over the place. And this brought me to the real insanity. Chrome has bugs too. Literal Hell Let’s take a concrete example: launch Chrome with `--disable-features=Vulkan` and check out this tiny SVG: [chromebug.html](#) This is most likely what you see: The reason you see this is because you are on a big, strong desktop GPU which doesn’t give a shit about load/store ops or uninitialized GPU memory. You’re driving a giant industrial bulldozer on your morning commute: traffic no longer exists and stop signals are fully optional. On a wimpy tiling GPU, however, things are different. Using a recent version of zink, even on a desktop GPU, you can run the same Chrome browser using `ZINK_DEBUG=rp,rploads` to enable the same codepaths used by tilers and also clear all uninitialized memory to red. Now load the same SVG, and you’ll see this: It took nearly a week of pair debugging and a new zink debug mode to prune down test cases and figure out what was happening. All around the composited SVG texture, memory is uninitialized. But this only shows up on tiling GPUs. And only if the driver is doing near-lethal amounts of very legal renderpass optimizations. This fast is too fast.

- [Alyssa Rosenzweig: Dissecting the Apple M1 GPU, the end](#) (2025/08/26 05:00)

In 2020, Apple released the M1 with a custom GPU. We got to work reverse-engineering the hardware and porting Linux. Today, you can run Linux on a range of M1 and M2 Macs, with almost all hardware working: wireless, audio, and full graphics acceleration. Our story begins in December 2020, when Hector Martin kicked off Asahi Linux. I was working for Collabora working on Panfrost, the open source Mesa3D driver for Arm Mali GPUs. Hector put out a public call for guidance from upstream open source maintainers, and I bit. I just intended to give some quick pointers. Instead, I bought myself a Christmas present and got to work. In between my university coursework and Collabora work, I poked at the shader instruction set. One thing led to another. Within a few weeks, I drew a triangle. In 3D graphics, once you can draw a triangle, you can do anything. Pretty soon, I started work on a shader compiler. After my final exams that semester, I took a few days off from Collabora to bring up an OpenGL driver capable of spinning gears with my new compiler. Over the next year, I kept reverse-engineering and improving the driver until it could run 3D games on macOS. Meanwhile, Asahi Lina wrote a kernel driver for the Apple GPU. My userspace OpenGL driver ran on macOS, leaving her kernel driver as the missing piece for an open source graphics stack. In December 2022, we shipped graphics acceleration in Asahi Linux. In January 2023, I started my final semester in my Computer Science program at the University of Toronto. For years I juggled my courses

with my part-time job and my hobby driver. I faced the same question as my peers: what will I do after graduation? Maybe Panfrost? I started reverse-engineering of the Mali Midgard GPU back in 2017, when I was still in high school. That led to an internship at Collabora in 2019 once I graduated, turning into my job throughout four years of university. During that time, Panfrost grew from a kid's pet project based on blackbox reverse-engineering, to a professional driver engineered by a team with Arm's backing and hardware documentation. I did what I set out to do, and the project succeeded beyond my dreams. It was time to move on. What did I want to do next? Finish what I started with the M1. Ship a great driver. Bring full, conformant OpenGL drivers to the M1. Apple's drivers are not conformant, but we should strive for the industry standard. Bring full, conformant Vulkan to Apple platforms, disproving the myth that Vulkan isn't suitable for Apple hardware. Bring Proton gaming to Asahi Linux. Thanks to Valve's work for the Steam Deck, Windows games can run better on Linux than even on Windows. Why not reap those benefits on the M1? Panfrost was my challenge until we "won". My next challenge? Gaming on Linux on M1. Once I finished my coursework, I started full-time on gaming on Linux. Within a month, we shipped OpenGL 3.1 on Asahi Linux. A few weeks later, we passed official conformance for OpenGL ES 3.1. That put us at feature parity with Panfrost. I wanted to go further. OpenGL (ES) 3.2 requires geometry shaders, a legacy feature not supported by either Arm or Apple hardware. The proprietary OpenGL drivers emulate geometry shaders with compute, but there was no open source prior art to borrow. Even though multiple Mesa drivers need geometry/tessellation emulation, nobody did the work to get there. My early progress on OpenGL was fast thanks to the mature common code in Mesa. It was time to pay it forward. Over the rest of the year, I implemented geometry/tessellation shader emulation. And also the rest of the owl. In January 2024, I passed conformance for the full OpenGL 4.6 specification, finishing up OpenGL. Vulkan wasn't too bad, either. I polished the OpenGL driver for a few months, but once I started typing a Vulkan driver, I passed 1.3 conformance in a few weeks. What remained was wiring up the geometry/tessellation emulation to my shiny new Vulkan driver, since those are required for Direct3D. Et voilà, Proton games. Along the way, Karol Herbst passed OpenCL 3.0 conformance on the M1, running my compiler atop his "rusticl" frontend. Meanwhile, when the Vulkan 1.4 specification was published, we were ready and shipped a conformant implementation on the same day. After that, I implemented sparse texture support, unlocking Direct3D 12 via Proton. ...Now what? Ship a great driver? Check. Conformant OpenGL 4.6, OpenGL ES 3.2, and OpenCL 3.0? Check. Conformant Vulkan 1.4? Check. Proton gaming? Check. That's a wrap. We've succeeded beyond my dreams. The challenges I chased, I have tackled. The drivers are fully upstream in Mesa. Performance isn't too bad. With the Vulkan on Apple myth busted, conformant Vulkan is now coming to macOS via LunarG's KosmicKrisp project building on my work. Satisfied, I am now stepping away from the Apple ecosystem. My friends in the Asahi Linux orbit will carry the torch from here. As for me? Onto the next challenge!

- [Sebastian Wick: Testing with Portals](#) (2025/08/21 21:00)

At the Linux App Summit (LAS) in Albania three months ago, I gave a talk about testing in the xdg-desktop-portal project. There is a recording of the presentation, and the slides are available as well. To give a quick summary of the work I did: Revamped the CI Reworked and improved the pytest based integration test harness Added integration tests for new portals Ported over all the existing GLib/C based integration tests Support ASAN for detecting memory leaks in the tests Made tests pretend to be either a host, Flatpak or Snap app The hope I had is that this will result in: Fewer regressions Tests for new features and bug fixes More confidence in refactoring More activity in the project While it's hard to get definite data on those points, at least some of it seems to have become reality. I have seen an increase in activity (there are other factors to this for sure), and a lot of PRs already come with tests without me even having to ask for it. Canonical is involved again, taking care of the Snap side of things. So far it seems like we didn't introduce any new regressions, but this usually shows after a new release. The experience of refactoring

portals also became a lot better because there is a baseline level of confidence when the tests pass, as well as the possibility to easily bisect issues. Overall I'm already quite happy with the results. Two weeks ago, Georges merged the last piece of what I talked about in the LAS presentation, so we're finally testing the code paths that are specific to host, Flatpak and Snap applications! I also continued a bit with improving the tests, and now they can be run with Valgrind, which is super slow and that's why we're not doing it in the CI, but it tends to find memory leaks which ASAN does not. With the existing tests, it found 9 small memory leaks. If you want to improve the Flatpak story, come and contribute to xdg-desktop-portal. It's now easier than ever!

From:

<https://wiki.tromjaro.alexio.tf/> - **TROMjaro wiki**

Permanent link:

<https://wiki.tromjaro.alexio.tf/doku.php?id=news:planet:freedesktop>

Last update: **2021/10/30 11:41**

