(How to avoid) Botching up ioctls

From: https://blog.ffwll.ch/2013/11/botching-up-ioctls.html

One clear insight kernel graphics hackers gained in the past few years is that trying to come up with a unified interface to manage the execution units and memory on completely different GPUs is a futile effort. So nowadays every driver has its own set of ioctls to allocate memory and submit work to the GPU. Which is nice, since there's no more insanity in the form of fake-generic, but actually only used once interfaces. But the clear downside is that there's much more potential to screw things up.

To avoid repeating all the same mistakes again I've written up some of the lessons learned while botching the job for the drm/i915 driver. Most of these only cover technicalities and not the big-picture issues like what the command submission ioctl exactly should look like. Learning these lessons is probably something every GPU driver has to do on its own.

Prerequisites

First the prerequisites. Without these you have already failed, because you will need to add a 32-bit compat layer:

- Only use fixed sized integers. To avoid conflicts with typedefs in userspace the kernel has special types like __u32, __s64. Use them.
- Align everything to the natural size and use explicit padding. 32-bit platforms don't necessarily align 64-bit values to 64-bit boundaries, but 64-bit platforms do. So we always need padding to the natural size to get this right.
- Pad the entire struct to a multiple of 64-bits if the structure contains 64-bit types - the structure size will otherwise differ on 32-bit versus 64-bit. Having a different structure size hurts when passing arrays of structures to the kernel, or if the kernel checks the structure size, which e.g. the drm core does.
- Pointers are __u64, cast from/to a uintprt_t on the userspace side and from/to a void __user * in the kernel. Try really hard not to delay this conversion or worse, fiddle the raw __u64 through your code since that diminishes the checking tools like sparse can provide. The macro u64_to_user_ptr can be used in the kernel to avoid warnings about integers and pointers of different sizes.

Basics

With the joys of writing a compat layer avoided we can take a look at the basic fumbles. Neglecting these will make backward and forward compatibility a real pain. And since getting things wrong on the first attempt is guaranteed you will have a second iteration or at least an extension for any given interface.

- Have a clear way for userspace to figure out whether your new ioctl or ioctl extension is supported on a given kernel. If you can't rely on old kernels rejecting the new flags/modes or ioctls (since doing that was botched in the past) then you need a driver feature flag or revision number somewhere.
- Have a plan for extending ioctls with new flags or new fields at the end of the structure. The drm core checks the passed-in size for each ioctl call and zero-extends any mismatches between kernel and userspace. That helps, but isn't a complete solution since newer userspace on older kernels won't notice that the newly added fields at the end get ignored. So this still needs a new driver feature flags.
- Check all unused fields and flags and all the padding for whether it's 0, and reject the ioctl if that's not the case. Otherwise your nice plan for future extensions is going right down the gutters since someone will submit an ioctl struct with random stack garbage in the yet unused parts. Which then bakes in the ABI that those fields can never be used for anything else but garbage. This is also the reason why you must explicitly pad all structures, even if you never use them in an array - the padding the compiler might insert could contain garbage.
- Have simple testcases for all of the above.

Fun with Error Paths

Nowadays we don't have any excuse left any more for drm drivers being neat little root exploits. This means we both need full input validation and solid error handling paths - GPUs will die eventually in the oddmost corner cases anyway:

- The ioctl must check for array overflows. Also it needs to check for over/underflows and clamping issues of integer values in general. The usual example is sprite positioning values fed directly into the hardware with the hardware just having 12 bits or so. Works nicely until some odd display server doesn't bother with clamping itself and the cursor wraps around the screen.
- Have simple testcases for every input validation failure case in your ioctl. Check that the error code matches your expectations. And finally make sure that you only test for one single error path in each subtest by submitting otherwise perfectly valid data. Without this an earlier check might reject the ioctl already and shadow the codepath you actually want to test, hiding bugs and regressions.
- Make all your ioctls restartable. First X really loves signals and second this will allow you to test 90% of all error

handling paths by just interrupting your main test suite constantly with signals. Thanks to X's love for signal you'll get an excellent base coverage of all your error paths pretty much for free for graphics drivers. Also, be consistent with how you handle ioctl restarting - e.g. drm has a tiny drmIoctl helper in its userspace library. The i915 driver botched this with the set_tiling ioctl, now we're stuck forever with some arcane semantics in both the kernel and userspace.
- If you can't make a given codepath restartable make a stuck task at least killable. GPUs just die and your users won't like you more if you hang their entire box (by means of an unkillable X process). If the state recovery is still too tricky have a timeout or hangcheck safety net as a last-ditch effort in case the hardware has gone bananas.
- Have testcases for the really tricky corner cases in your error recovery code - it's way too easy to create a deadlock between your hangcheck code and waiters.

Time, Waiting and Missing it

GPUs do most everything asynchronously, so we have a need to time operations and wait for outstanding ones. This is really tricky business; at the moment none of the ioctls supported by the drm/i915 get this fully right, which means there's still tons more lessons to learn here.

- Use CLOCK_MONOTONIC as your reference time, always. It's what alsa, drm and v4l use by default nowadays. But let userspace know which timestamps are derived from different clock domains like your main system clock (provided by the kernel) or some independent hardware counter somewhere else. Clocks will mismatch if you look close enough, but if performance measuring tools have this information they can at least compensate. If your userspace can get at the raw values of some clocks (e.g. through in-command-stream performance counter sampling instructions) consider exposing those also.
- Use __s64 seconds plus __u64 nanoseconds to specify time. It's not the most convenient time specification, but it's mostly the standard.
- Check that input time values are normalized and reject them if not. Note that the kernel native struct ktime has a signed integer for both seconds and nanoseconds, so beware here.
- For timeouts, use absolute times. If you're a good fellow and made your ioctl restartable relative timeouts tend to be too coarse and can indefinitely extend your wait time due to rounding on each restart. Especially if your reference clock is something really slow like the display frame counter. With a spec lawyer hat on this isn't a bug since timeouts can always be extended - but users will surely hate you if their neat animations starts to stutter due to this.
- Consider ditching any synchronous wait ioctls with timeouts and just deliver an asynchronous event on a pollable file descriptor. It fits much better into event driven applications' main loop.
- Have testcases for corner-cases, especially whether the return values for already-completed events, successful waits and timed-out waits are all sane and suiting to your needs.

Leaking Resources, Not

A full-blown drm driver essentially implements a little OS, but specialized to the given GPU platforms. This means a driver needs to expose tons of handles for different objects and other resources to userspace. Doing that right entails its own little set of pitfalls:

- Always attach the lifetime of your dynamically created resources to the lifetime of a file descriptor. Consider using a 1:1 mapping if your resource needs to be shared across processes - fd-passing over unix domain sockets also simplifies lifetime management for userspace.
- Always have O_CLOEXEC support.
- Ensure that you have sufficient insulation between different clients. By default pick a private per-fd namespace which forces any sharing to be done explicitly. Only go with a more global per-device namespace if the objects are truly device-unique. One counterexample in the drm modeset interfaces is that the per-device modeset objects like connectors share a namespace with framebuffer objects, which mostly are not shared at all. A separate namespace, private by default, for framebuffers would have been more suitable.
- Think about uniqueness requirements for userspace handles. E.g. for most drm drivers it's a userspace bug to submit the same object twice in the same command submission ioctl. But then if objects are shareable userspace needs to know whether it has seen an imported object from a different process already or not. I haven't tried this myself yet due to lack of a new class of objects, but consider using inode numbers on your shared file descriptors as unique identifiers - it's how real files are told apart, too. Unfortunately this requires a full-blown virtual filesystem in the kernel.

Last, but not Least

Not every problem needs a new ioctl:

- Think hard whether you really want a driver-private interface. Of course it's much quicker to push a driver-private interface than engaging in lengthy discussions for a more generic solution. And occasionally doing a private interface to spearhead a new concept is what's required. But in the end, once the generic interface comes around you'll end up maintainer two interfaces. Indefinitely.
- Consider other interfaces than ioctls. A sysfs attribute is much better for per-device settings, or for child objects

with fairly static lifetimes (like output connectors in drm with all the detection override attributes). Or maybe only your testsuite needs this interface, and then debugfs with its disclaimer of not having a stable ABI would be better.

Finally, the name of the game is to get it right on the first attempt, since if your driver proves popular and your hardware platforms long-lived then you'll be stuck with a given ioctl essentially forever. You can try to deprecate horrible ioctls on newer iterations of your hardware, but generally it takes years to accomplish this. And then again years until the last user able to complain about regressions disappears, too.