



Why we can't have nice things

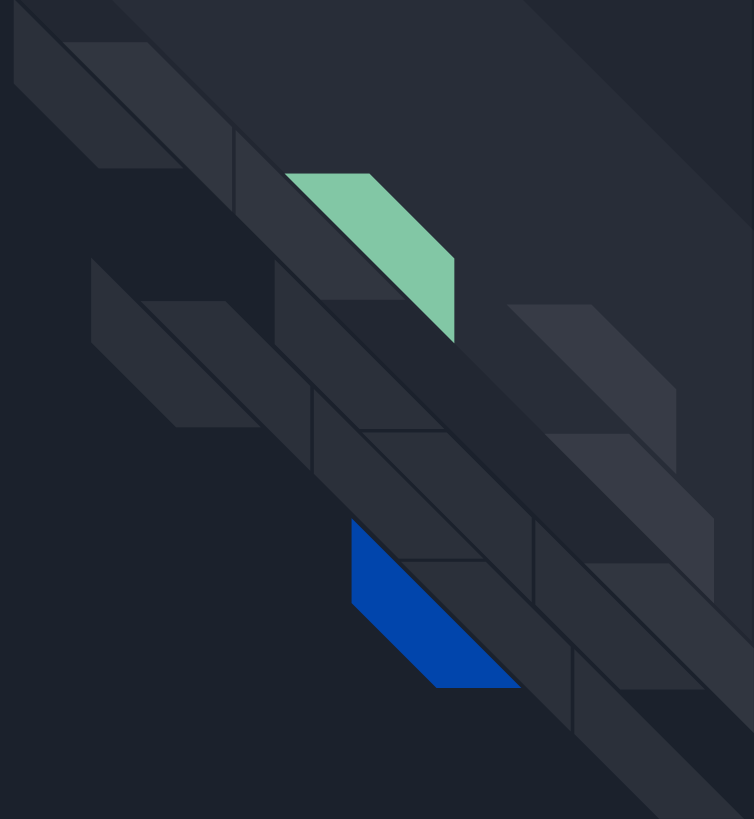
The beautiful future of userspace-controlled synchronization and why it's not possible

Jason Ekstrand, LPC 2021



Special blame

Current status of GPU synchronization in Linux





We have a *lot* of synchronization primitives

Linux kernel:

- struct dma_fence
- struct dma_resv
- struct sync_file
- struct drm_syncobj
- struct drm_syncobj (timeline)

X11:

- xshmfence

Wayland:

- wl_callback

And many more....

OpenGL/EGL:

- glFinish()
- glWaitSync()
- glReadPixels()
- Many other GL calls, implicitly

Vulkan:

- VkFence
- VkSemaphore
- VkSemaphore (timeline)

In the kernel, everything
is struct dma_fence





What is struct dma_fence?

A struct which represents a (potentially future) event:

- Has a boolean “signaled” state
- Has a bunch of useful utility helpers/concepts:
 - Reference-counted
 - Callback-based wait mechanism
 - Lazy CPU-signal binding (good for GPU <-> GPU sync inside a driver)

Provides two very useful guarantees:

- One-shot: once signaled, will be signaled forever
- Finite-time: once exposed, is guaranteed signal in a reasonable amount of time



Finite-time guarantee of dma_fence

The finite-time guarantee for dma_fence has several important implications:

- Cannot have circular dependencies
- Nothing which is *required* for signaling a dma_fence may fail
 - Most locks not allowed on the dma_fence signal path
 - Memory allocation must be GFP_NOWAIT/GFP_ATOMIC
- If the GPU or other HW hangs:
 - Reset the chip
 - Kill the userspace connection
 - Signal all associated fences
- You can wait on a struct dma_fence in kernel-space
 - May block swapping, BO migration, the shrinker, etc.

This is really nice for the kernel!

Userspace wants control





AMD and Intel want to side-step the kernel

Intel has plans for a direct-to-firmware submit model:

- Kernel still manages memory and global resources
- Userspace tells the kernel which resources should be resident
- Userspace submits batches directly to firmware
- In theory, this should be faster and lower-latency

AMD and Arm have expressed similar plans for their future GPUs

Intel has an emulation of this called ULLS, being used for compute today



High-performance clients want timelines

Vulkan recently added the `VK_KHR_timeline_semaphore` extension:

- Each semaphore is a single 64-bit integer value
- Signaling sets a higher value (must increase)
- Waiting waits on the value to be \geq target
- Supports CPU and GPU signal/wait
- Replaces both old-school `VkSemaphore` and `VkFence`

This looks a lot like what we all do inside our drivers....

It's the same model game developers get in D3D12 via Monitored Fence on Windows 10

- Among other things, it's a way better model for multi-threaded engines



High-performance clients want timelines

Vulkan timeline semaphores come with some caveats:

- Naturally supports wait-before-signal
- Clients can deadlock themselves
 - Client gets to keep the pieces if this happens

We've emulated timeline semaphores using timeline struct `drm_syncobj`:

- Userspace driver has a thread for managing outstanding requests
- Batches aren't submitted to the kernel until all dependencies are resolved
- Any deadlocks stay in userspace

This really isn't as efficient as we'd like...



Compute doesn't happen in finite time

For 3D, almost everything completes in $< 1s$


- Typical monitor refresh is 60 Hz
- Games get unplayable below 20 Hz

In compute, this isn't true

- The GPU is just another processor, with a giant pile of cores
- Simulations often run for hours, days, or weeks
- Why shouldn't a kernel run on the GPU for 3 days?
- You can't use `struct dma_fence` for "Is my compute job done yet?"

The glorious future!





One possible model: Userspace Memory Fence (UMF)

Basically, expose the common seqno concept directly to userspace:

- Store a 64 or 32-bit value in CPU mappable memory
 - Windows requires it to live in system memory, maybe we should too?
- Signaling is done by writing to memory and maybe signaling an interrupt
 - Might be done by the kernel as part of the exec ioctl
 - Userspace may want to do this itself!
- CPU waits are done with an ioctl similar to futex()
 - Maybe we can just use futex()?
- For GPU waits, the exec ioctl takes a pointer and a target value
 - Maybe also a configurable comparison operator?

This would let us implement timeline semaphores directly!



Other possible models

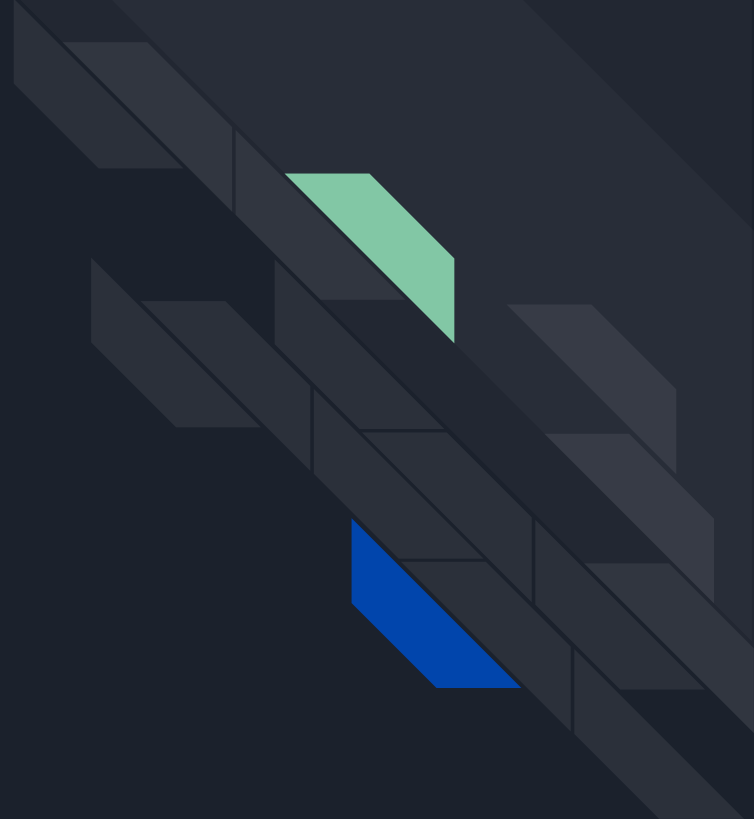
Other possible models exist but I won't enumerate them all here

... because they all have the same problems. 😞

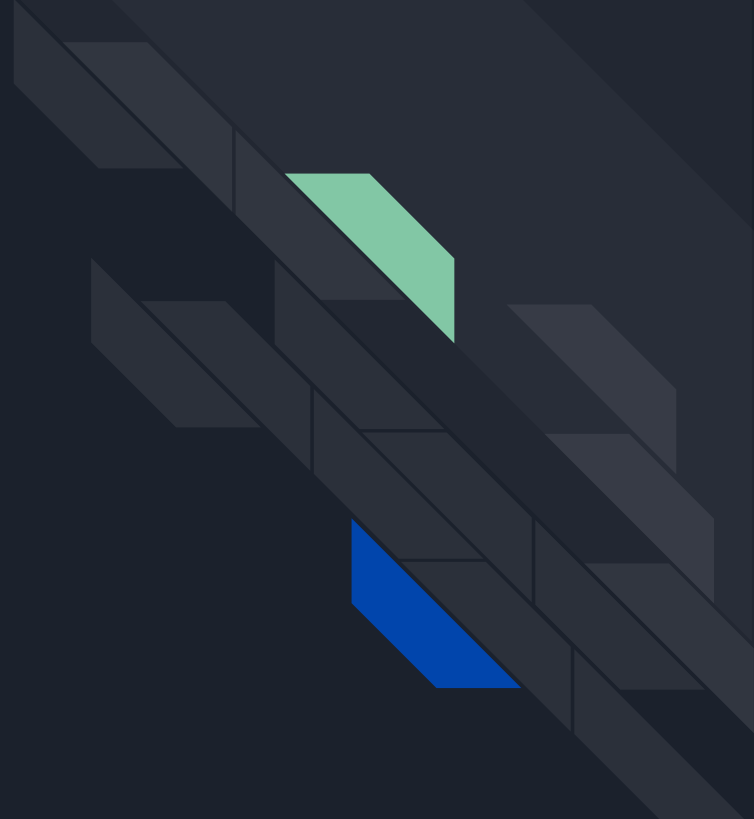
The actual future



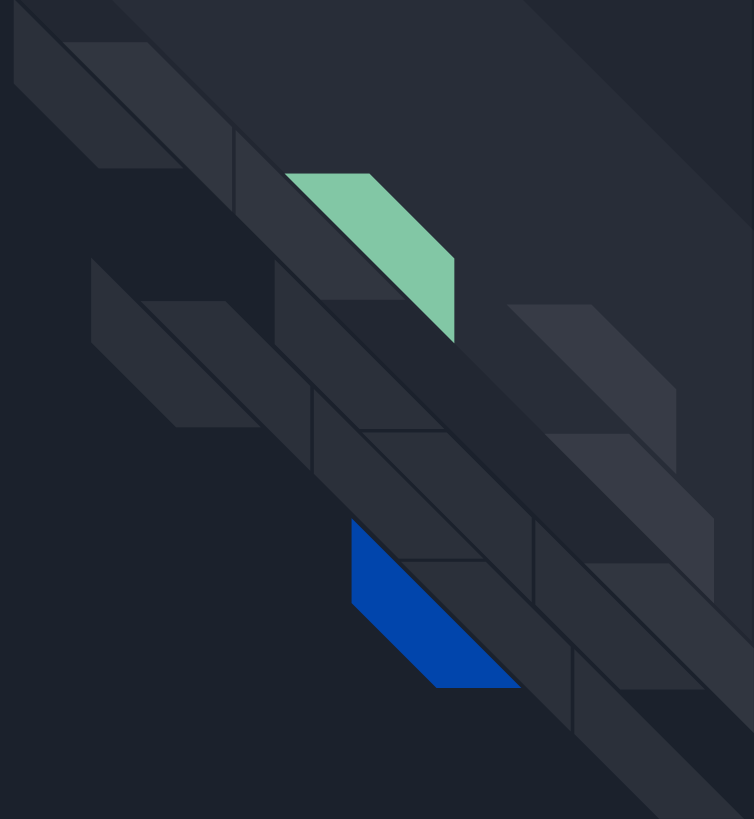
Why so bleak?



None of userspace
works this way



Why not wrap it in a
struct dma_fence?





We can't trust userspace

No more “finite time” guarantee

- Userspace may never bother to signal it
- Userspace might signal it wrong
- Userspace may deadlock


Ok, fine, so throw a timeout on it. That should work, right?.....



Userspace can't trust us

Let's assume userspace submits a bunch of jobs with an acyclic dependency graph

- We don't know what that graph looks like
- We might want to move some memory around
- That adds dependencies to the graph
- Now there's a cycle
- So a fence times out and we kill the userspace context
- But userspace didn't do anything wrong!



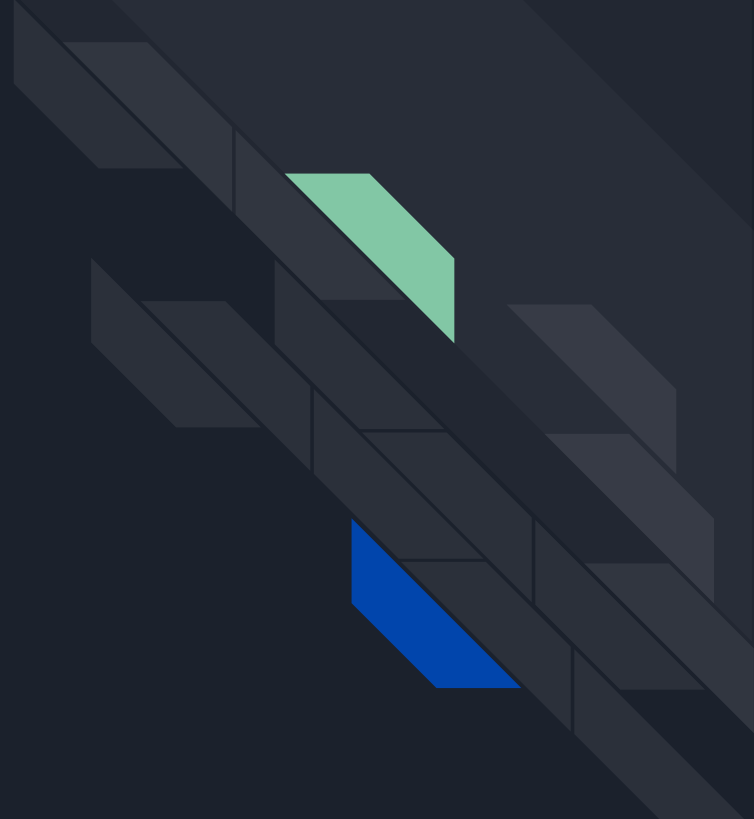
Solution pt. 1: Separate memory and execution synchronization

This is what Intel has prototyped to get some stuff working:

- Memory management happens separately from execution synchronization
- Internal kernel dependency tracking happens via struct `dma_fence`
 - TTM etc. use it everywhere
- Sync with userspace happens via userspace memory fences
- Internal kernel stuff *never* waits on a UMF!
- If you need to move memory, you preempt, move, restore
- Preempt happens in finite time so you can block `dma_fence` on it

Depends on preemption, but all the big GPUs can do that these days

Why don't you just do that, then?





We still want to interoperate

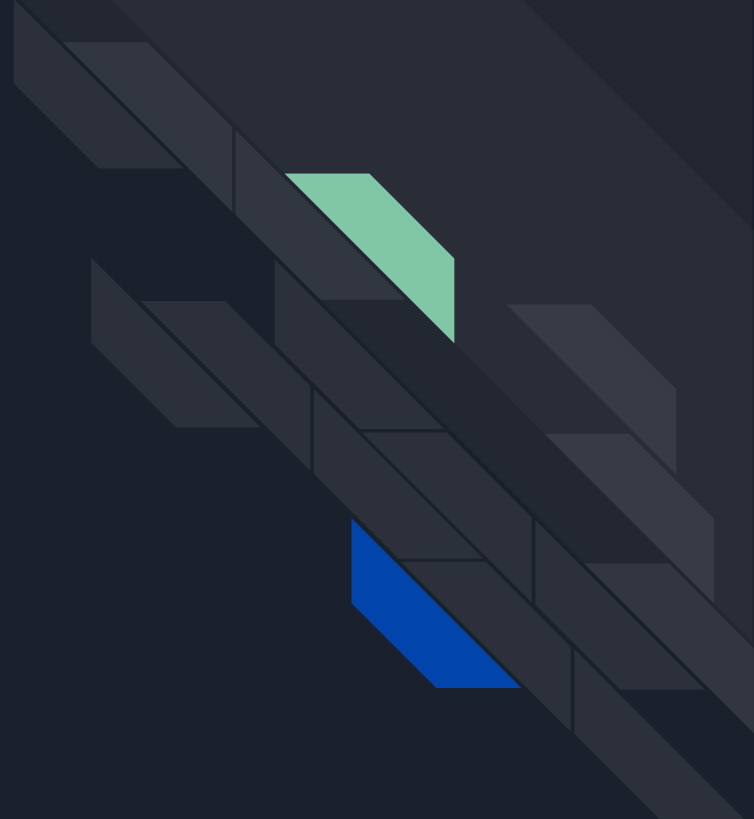
Our 3D drivers need to talk to X11, Wayland, Android, etc:

- X11 and (old) Wayland use implicit sync (struct dma_fence in the BO)
- Modern Wayland and Android use sync file (wrapper around struct dma_fence)
- EGL, CL \leftrightarrow GL interop, etc. all require implicit sync
- ...

If a job waits on a userspace fence, it cannot be used to signal a struct dma_fence!

What are our options?

Spoiler: They all suck!





Options?

Option 1: Re-plumb everything to support UMF

- All kernel drivers need strict memory/execution sync separation
- Need a new userspace-facing sync primitive to pass around
- All userspace drivers, compositors, etc. need updating

Option 2: Wait in userspace before passing off to another process

- Neatly solves all the deadlock problems
- No more cross-process pipelining -> higher latency
- Totally breaks wayland

Option 3: Be much more clever than me?



Suggestions?



Inside the kernel, there's only one

Everything is a struct `dma_fence`:

- struct `dma_resv` is a container of `dma_fences`
 - One exclusive fence or many shared fences
- struct `sync_file` is a container for turning a `dma_fence` into a file
 - Supports `poll()` to wait on the fence
- struct `drm_syncobj` is just a `dma_fence*` and a lock
 - Syncobj ioctls allow userspace to modify the pointer (not the `dma_fence`)
- Timeline struct `dma_syncobj` is a `syncobj` where the fence is a struct `dma_fence_chain`
 - Singly linked list of fences with associated timeline values
 - Automatically prunes signaled fences so it doesn't grow unbounded