Enabling user space drivers to vend more granular device access to client processes

Alex Mastro

Who we are

ASIC Platform Software

- Firmware
- Device drivers <= this talk
- Low-level host libraries
- Tools
- Test infra
- Device simulation

Meta

Agenda

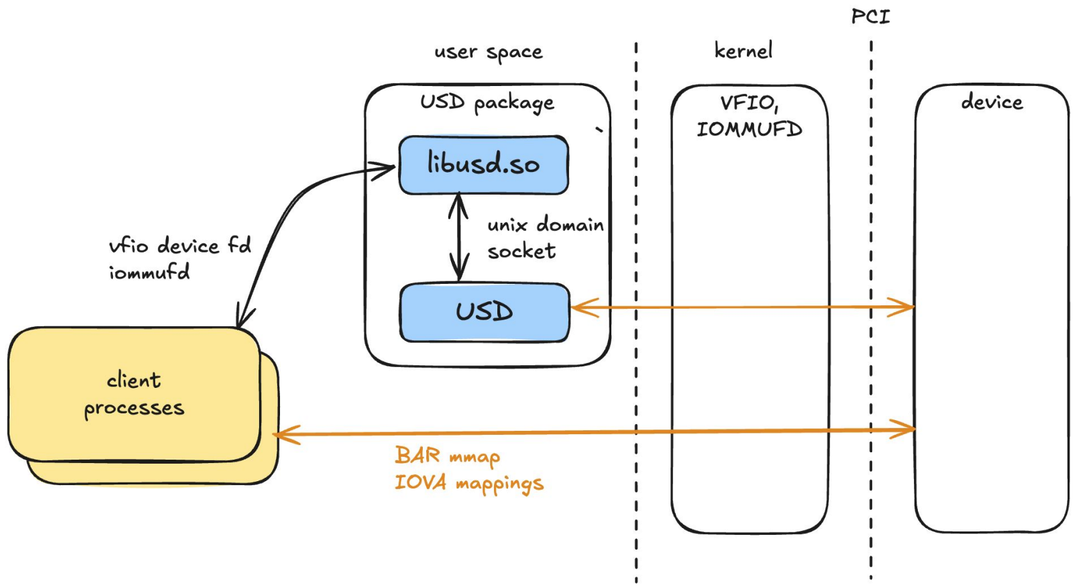
01 What we’re currently doing (and why)

02 Where we want to go

03 How can we get there?

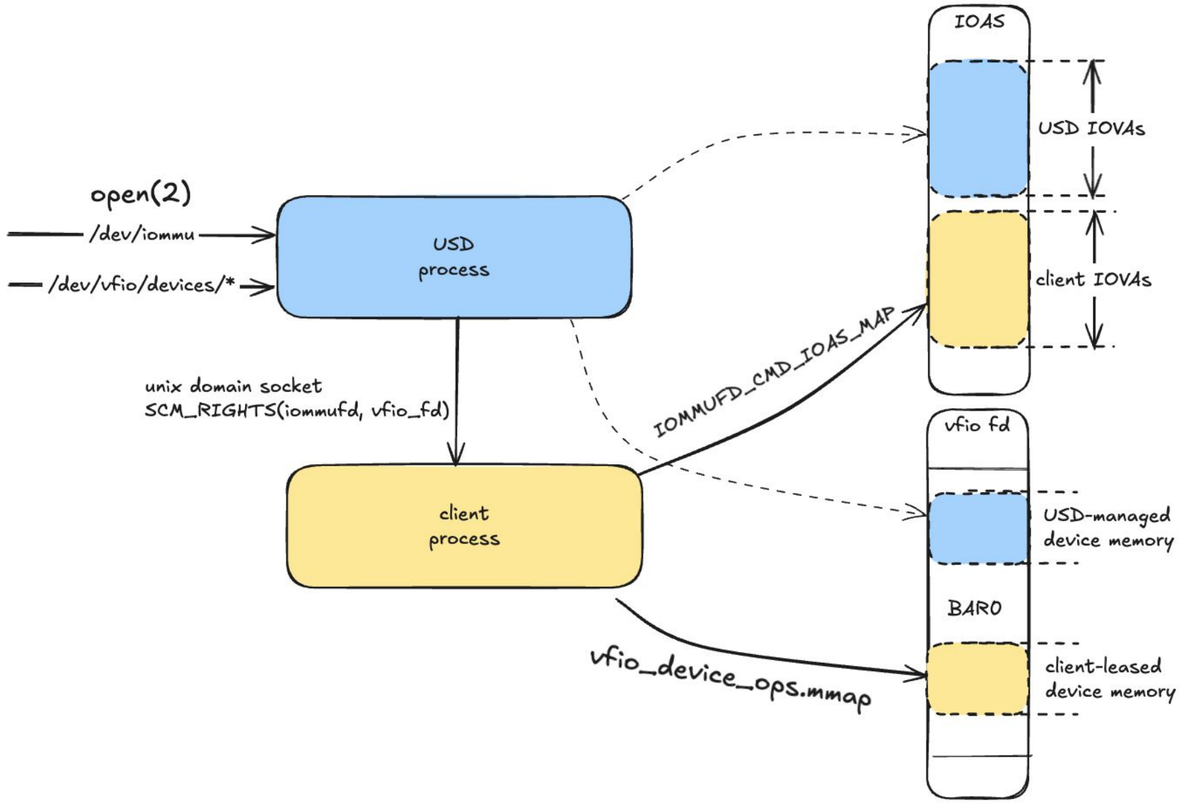
Current architecture

- User space driver (USD), firmware and supporting libraries are co-deployed in a single package
- USD owns traditional device driver aspects of device management (e.g. firmware bootstrap, resource arbitration, metrics collection, fault-handling, reset)
- Clients receive dups of vfio device fd, iommufd over unix sockets via SCM_RIGHTS
- Cooperative usage of the device's BARs and I/O address space (IOAS) between USD and clients



Why?

- Minimize overhead on top of PCI round trips (~µs)
- Paths through USD via IPC are too costly.
- Bounce client-initiated access to device memory via IPC => USD?
- Bounce device DMAs through USD to client?
- USD could vend IOMMU-mapped memfds to the client. Restricts API shape. Clients can’t bring their own memory to be pinned + mapped.
- Or… just let clients directly mmap device BARs, and IOMMU-map their own memory.



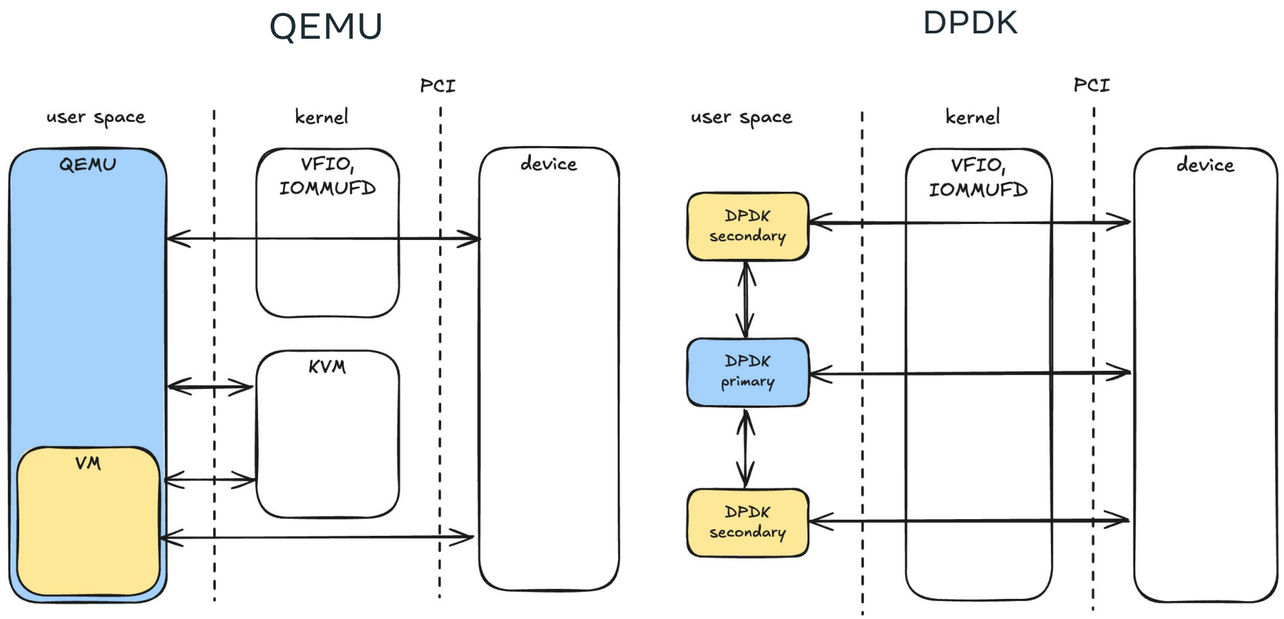
Other VFIO, IOMMUFD users?

QEMU

Device passthrough: a single process (qemu) orchestrates memory access between the VM and device(s) via VFIO, IOMMUFD, and KVM. No need to share direct device access to peer processes.

DPDK

Potentially multiple DPDK processes cooperatively access the device. Primary process shares VFIO fds with secondary processes via Unix domain sockets. All have symmetric access to the device HW. Fully cooperative. DPDK primary IOMMU-maps shared memory on behalf of peers.



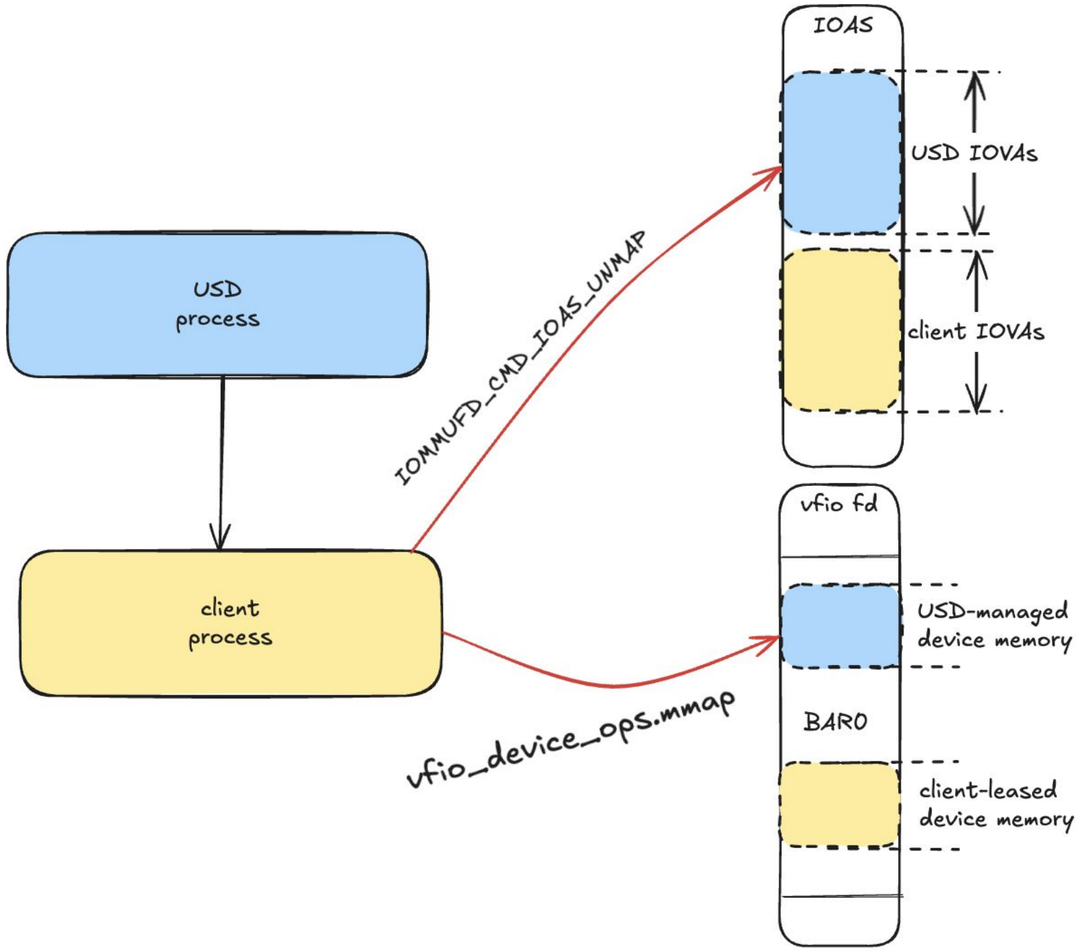
Downsides of current approach

More access is granted to client processes than is strictly required: they have full access to VFIO and IOMMUFD.

Their misbehavior could break other processes using the device (or the device itself)

- Stomp on parts of BARs they shouldn't (fundamental MMIOs, config space)
- Reset the device
- Unmap other processes IOVAs

Assumes complete trust between client processes and USD.



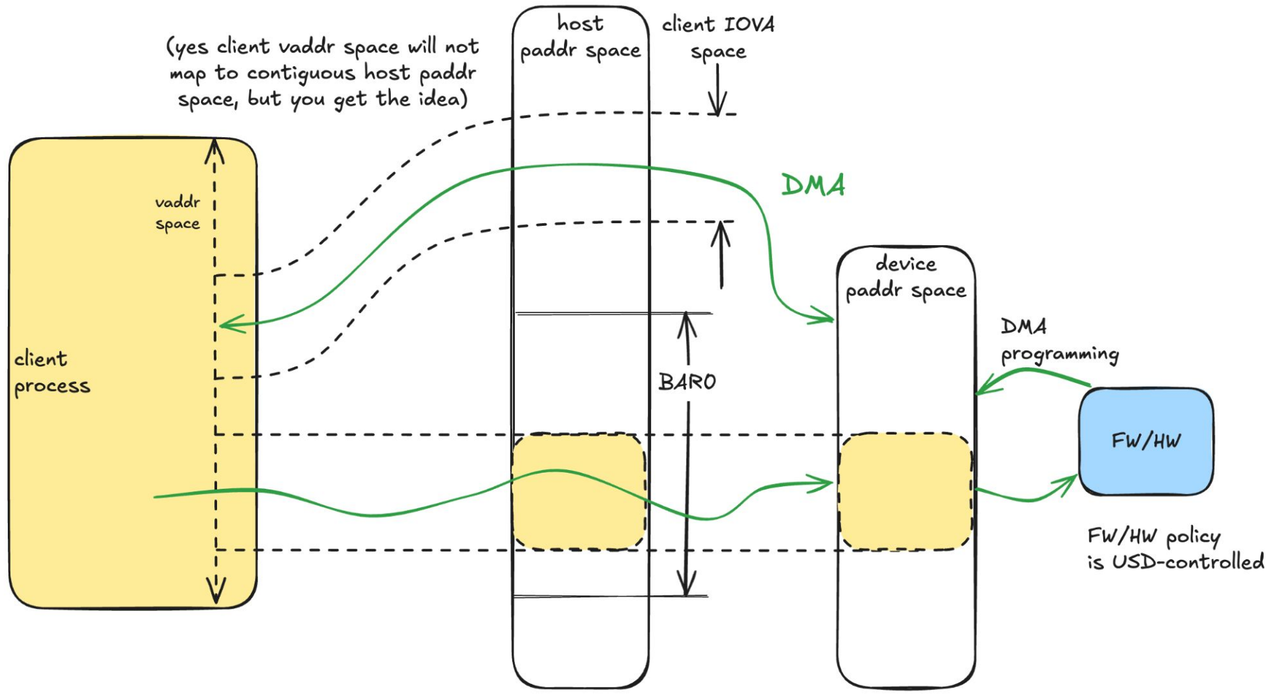
If the client can initiate DMAs, why bother with any of this?

If clients have the ability to initiate DMAs, couldn't they corrupt peer clients, or USD's IOMMU-mapped address space due to lack of isolation?

Client is only provided a view of device memory which USD grants.

The combination of USD and device FW/HW policy is responsible for restricting the side-effects which could arise from client-access to the externally-exposed device address ranges.

e.g. "DMA-commands received over submission queues belonging to client X are only allowed to target IOVA range Y"



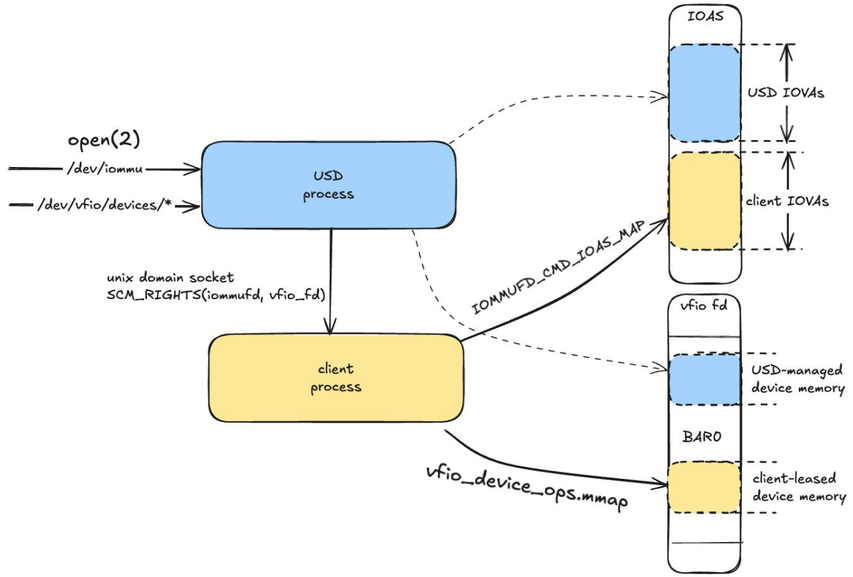
Initial goals

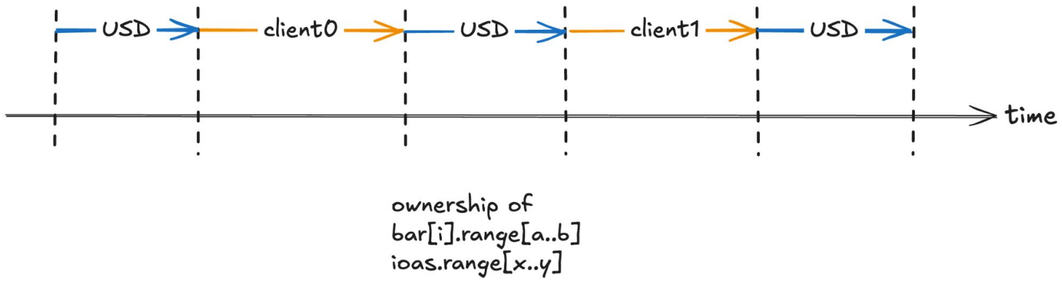
Augment kernel interfaces to give USDs the following abilities

- Enforce access/visibility hygiene: vend access which is strictly necessary for client <> device interaction
- Control and/or observe the lifetimes of the above after they have been vended to clients.

Non-goals

- Adopt a security posture which treats clients as completely untrusted. They need to mmap certain parts of BARs after all.
- Enable arbitrary nesting of device access levels (clients aren't granting further-restricted access to sub-clients)





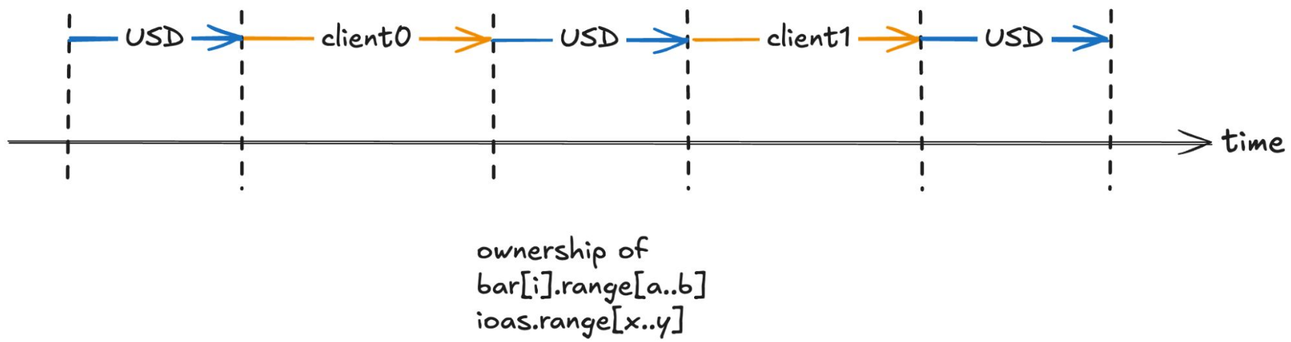
Lifetime control

USD vends clients the capability to create BAR and IOVA mappings.

It needs the tools to control and/or observe the lifetimes of that access.

Control: e.g. forcibly revoke access to BARs.

Observe: e.g. passively wait until outstanding usage has ceased before re-leasing is permitted.



Simplistic lifetime observation

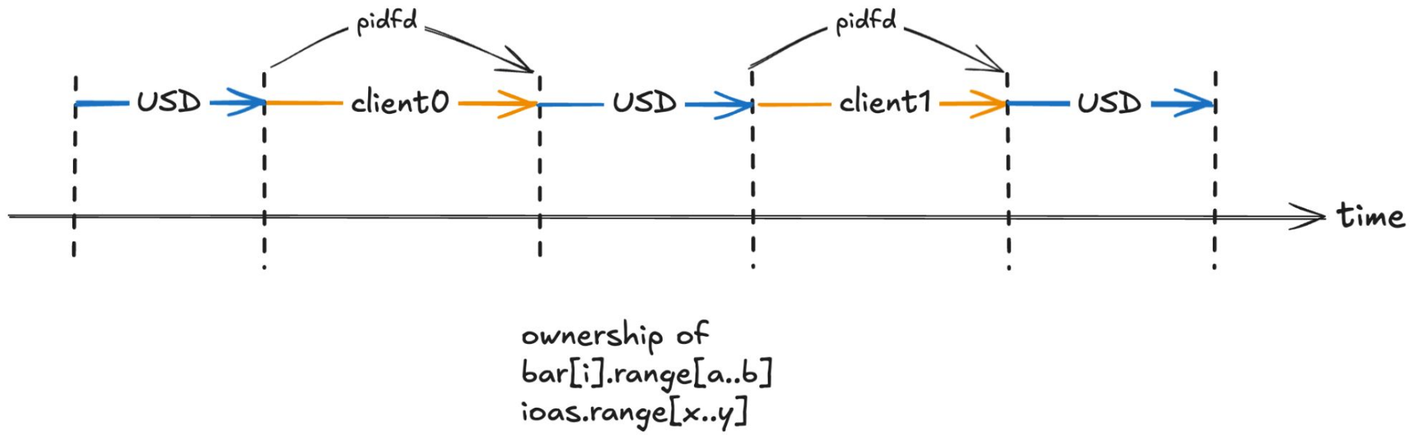
Use proxy information to infer whether access-conveying mechanisms have been released by clients.

e.g.

1. USD derives client pidfd after socket accept() via SO_PEERPIDFD
2. Uses it to detect when client process has exited, and infers that client-leased BAR, IOVA mappings are no longer in use.

Still relies on assumptions of trust/collaboration

- Client process lifetime is a superset of that of the device-access-conveying fds
- Client has *not* shared the fds with other processes via SCM_RIGHTS



VFIO ❤️ dma-buf!

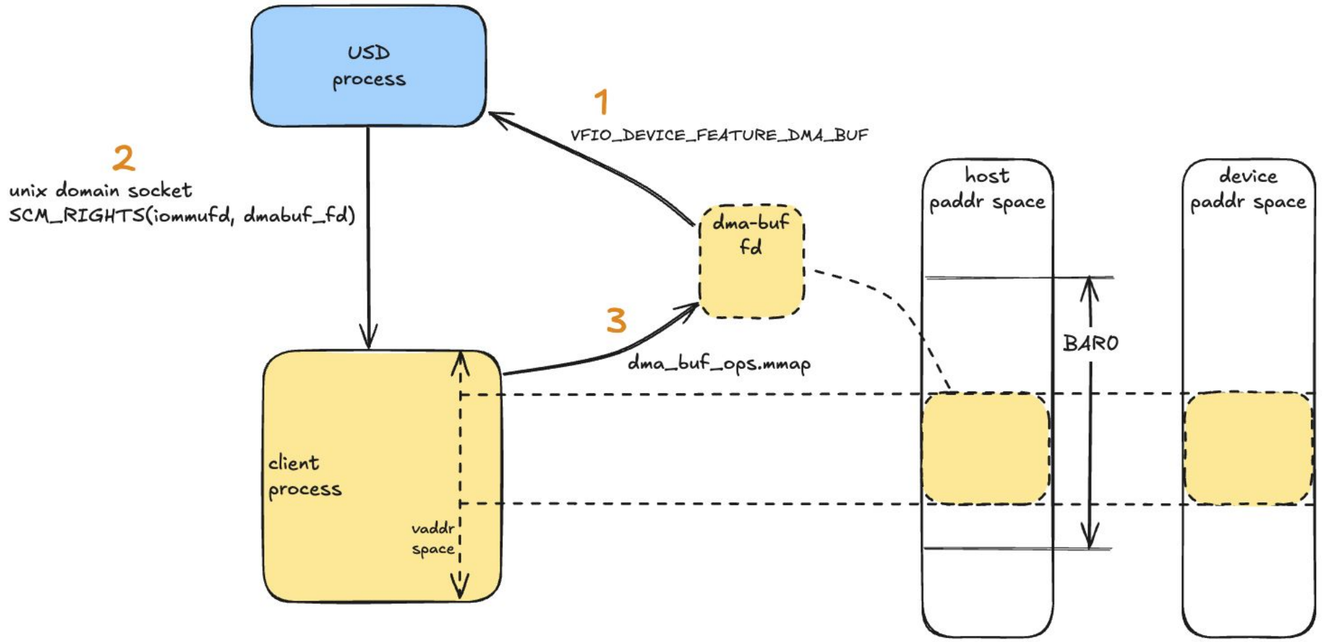
Just merged into linux-vfio 🎉 thanks Leon R.

VFIO_DEVICE_FEATURE_DMA_BUF gives us a fd backed by a kernel object which represents slices within a BAR for a given device.

Initial goal is to enable P2P, but can be naturally extended for our purposes. Just need dma_buf_ops.mmap().

Moving beyond simplistic lifetime control requires at least one of:

- USD initiated dma-buf revocation
- USD notification of dma_buf_ops.released()

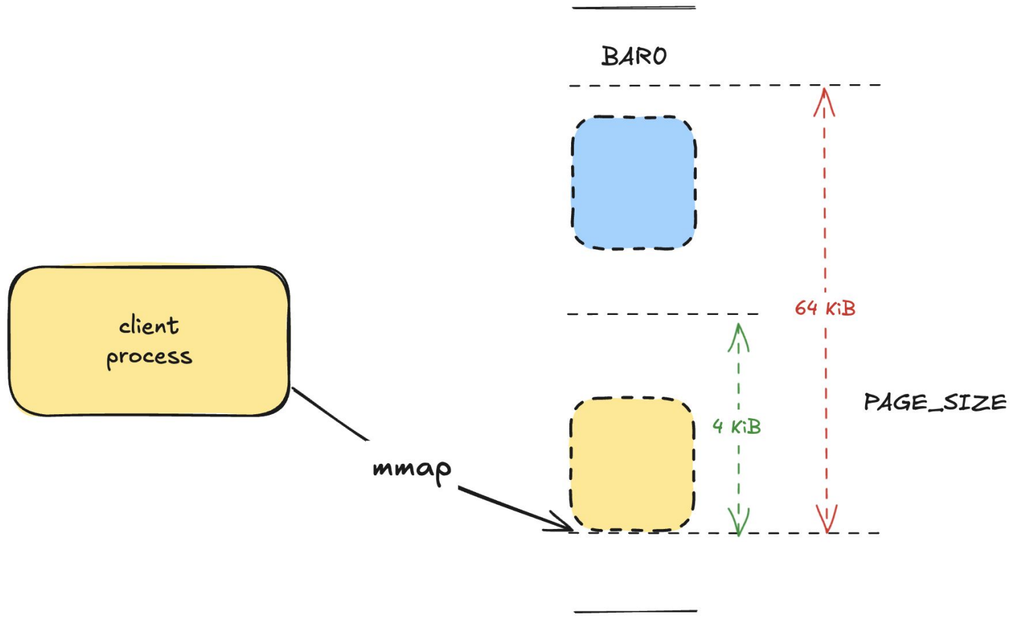


Device memory layout constraints

Minimum unit of device BAR memory exposure via mmap is PAGE_SIZE.

Device HW needs to align/stride per-process MMIO by at least PAGE_SIZE in order to maintain isolation.

Worst-case 64 KiB.



IOMMUFD access granularity

Client only needs to IOMMU_IOAS_{MAP,UNMAP} against specific IOVA ranges.

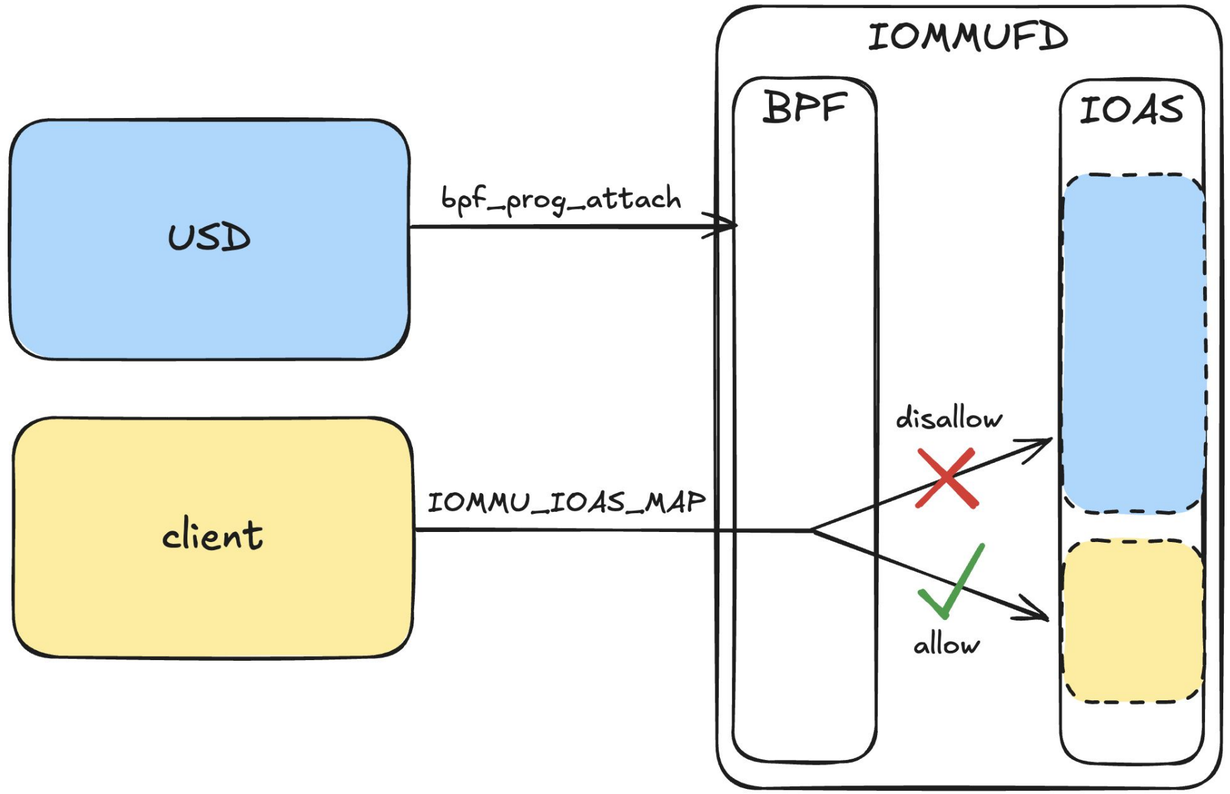
Avoid putting the kernel in a position where it needs to maintain arbitrarily-dimensioned access policies.

One idea: USD-attached BPF program implements access policy. Perhaps at IOMMUFD ioctl surface.

BPF program needs to be supplied with sufficient context to make an access verdict.

- ioctl num, user args
- caller context (e.g. pid, uid, gid)

BPF program execution should be scoped to relevant FD access only. Do not want to control access globally (e.g. for all ioctl syscalls)



Is the problem worth solving?

We think so.

Access hygiene is a property of robust systems.

These capabilities would narrow the access-control gap between in-kernel drivers and USDs.

Implementation line of sight?

VFIO + dma-buf, yes.
IOMMUFD + BPF is less clear.

Future composability?

To what extent would adding these features encumber VFIO / IOMMUFD's ability to pursue future, unrelated features?

VFIO + dma-buf seems like a natural fit. Proposed semantics have various forms of precedence.

IOMMUFD + BPF perhaps plausibly OK.

Thank you!