An Introduction to PCI Device Assignment with VFIO

Alex Williamson / alex.williamson@redhat.com

What is VFIO?

Officially:

Virtual Function I/O

Virtual Function I/O

...but it's not limited to SR-IOV, or even PCI

Some suggest...

Very Fast I/O

Very Fast I/O

Sort of, yeah...

I've also heard...

Virtual Fabric I/O

Virtual Fabric I/O

Fabric?

Let me propose...

Versatile Framework for userspace I/O

Versatile Framework for userspace I/O

(OK, not really, but it's more accurate)

VFIO is a secure, userspace driver framework

VFIO is a secure, userspace driver framework

- Hardware IOMMU based DMA mapping and isolation
  - IOMMU group based

VFIO is a secure, userspace driver framework

- Hardware IOMMU based DMA mapping and isolation
  - IOMMU group based
- Modular IOMMU and bus driver support
  - PCI and platform devices currently supported
  - IOMMU API (type1) and ppc64 (SPAPR) models

VFIO is a secure, userspace driver framework

- Hardware IOMMU based DMA mapping and isolation
  - IOMMU group based
- Modular IOMMU and bus driver support
  - PCI and platform devices currently supported
  - IOMMU API (type1) and ppc64 (SPAPR) models
- Full device access, DMA, and interrupt support
  - Read/write & mmap support of device resources
  - Mapping of user memory to I/O virtual addresses
  - eventfd and irqfd based signaling mechanisms

Userspace drivers?

Weren't we talking about device assignment...

The requirements are the same

- Access to device resources
- Isolation and secure DMA mapping through an IOMMU
- Interrupt signaling support

Device assignment is simply a multi-layer userspace driver

Also enables other userspace drivers

- **DPDK** - Data Plane Development Kit (NFV)
- **UNVMe** - A userspace NVMe driver
- **rVFIO** - Ruby wrapper gem for VFIO

How does VFIO work?

VFIO provides access to a device within a secure and programmable IOMMU context

Let's start with the device

Let's start with the device

- How does a driver program a device?

Let's start with the device

- How does a driver program a device?
- How does a device signal the driver?

Let's start with the device

- How does a driver program a device?
- How does a device signal the driver?
- How does a device transfer data?

Let's start with the device

- How does a driver program a device?
- How does a device signal the driver?
- How does a device transfer data?

VFIO takes an abstract view of a device, we want to support ***anything***

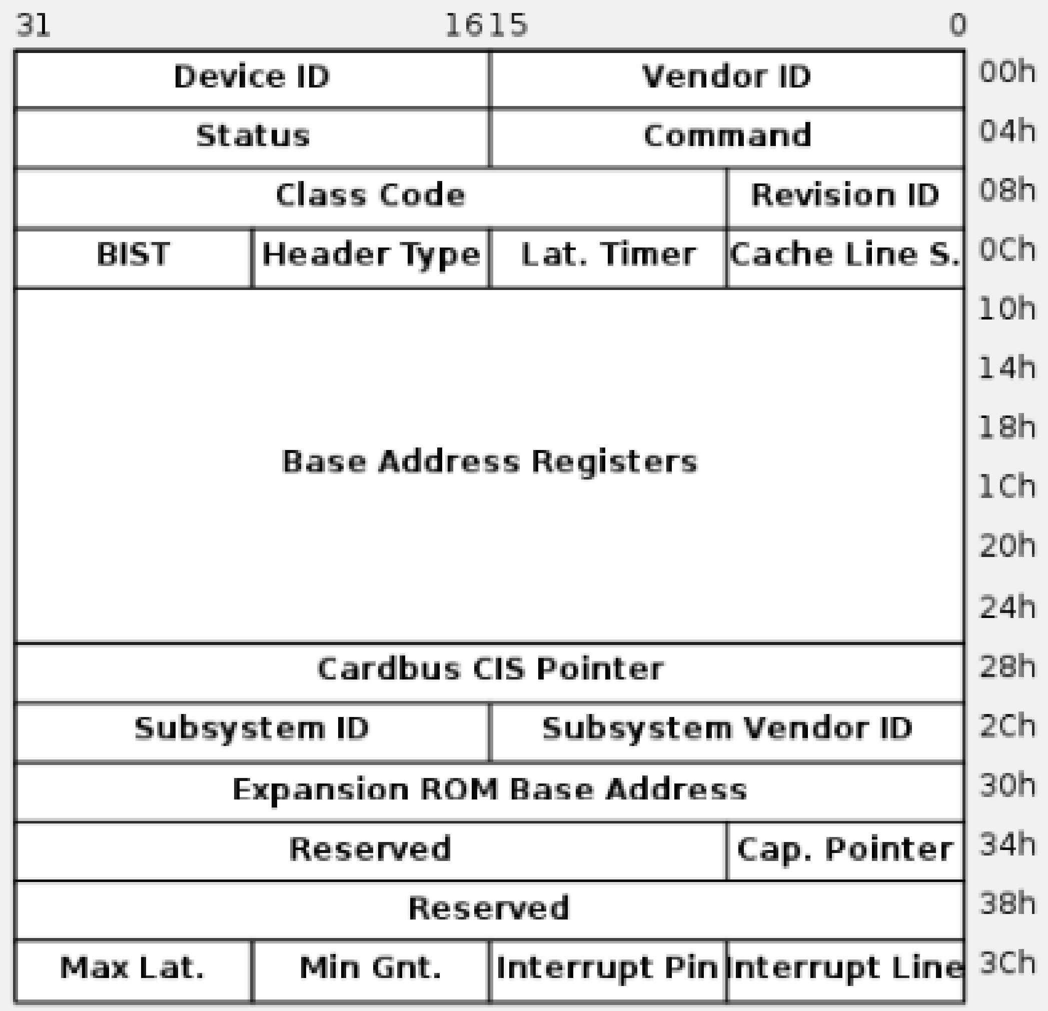
How does a device driver program a PCI device?

How does a device driver program a PCI device?

- Programmed I/O
  - IN/OUT
  - read/write

How does a device driver program a PCI device?

- Programmed I/O
  - IN/OUT
  - read/write
- PCI Configuration Space

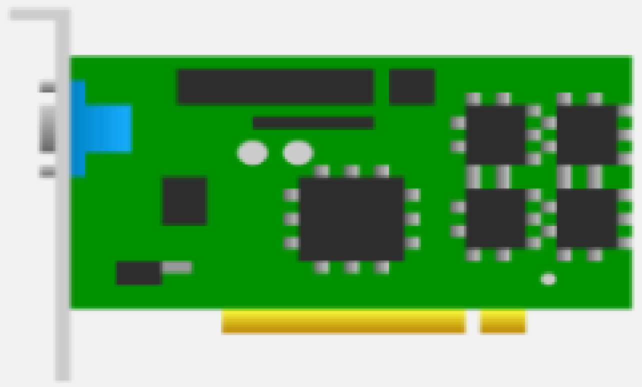


31	16	15	0	
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Revision ID	08h
BIST	Header Type	Lat. Timer	Cache Line S.	0Ch
Base Address Registers				10h
				14h
				18h
				1Ch
				20h
				24h
Cardbus CIS Pointer				28h
Subsystem ID		Subsystem Vendor ID		2Ch
Expansion ROM Base Address				30h
Reserved			Cap. Pointer	34h
Reserved				38h
Max Lat.	Min Gnt.	Interrupt Pin	Interrupt Line	3Ch

The VFIO device file descriptor

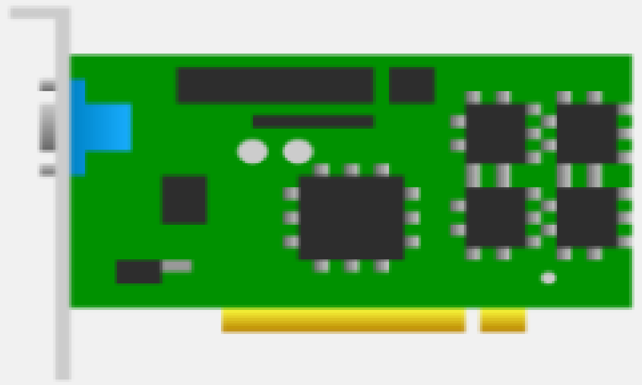
- Divided into regions
- Each region maps to a device resource
  - Ex. MMIO BAR, IO BAR, PCI config space
- Region count and info discovered through ioctl
  - File offset, allowable access, etc.

A PCI device example



```
01:00.0 VGA compatible controller: NVIDIA Corporation GM107
 Region 0: Memory at f6000000 (32-bit, non-prefetchable) [size=16M]
 Region 1: Memory at e0000000 (64-bit, prefetchable) [size=256M]
 Region 3: Memory at f0000000 (64-bit, prefetchable) [size=32M]
 Region 5: I/O ports at e000 [size=128]
 Expansion ROM at f7000000 [disabled] [size=512K]
```

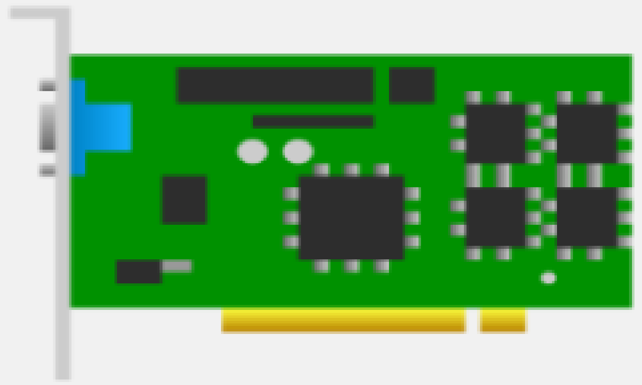
A PCI device example



```
01:00.0 VGA compatible controller: NVIDIA Corporation GM107
 Region 0: Memory at f6000000 (32-bit, non-prefetchable) [size=16M]
 Region 1: Memory at e0000000 (64-bit, prefetchable) [size=256M]
 Region 3: Memory at f0000000 (64-bit, prefetchable) [size=32M]
 Region 5: I/O ports at e000 [size=128]
 Expansion ROM at f7000000 [disabled] [size=512K]
```

These are all regions

A PCI device example



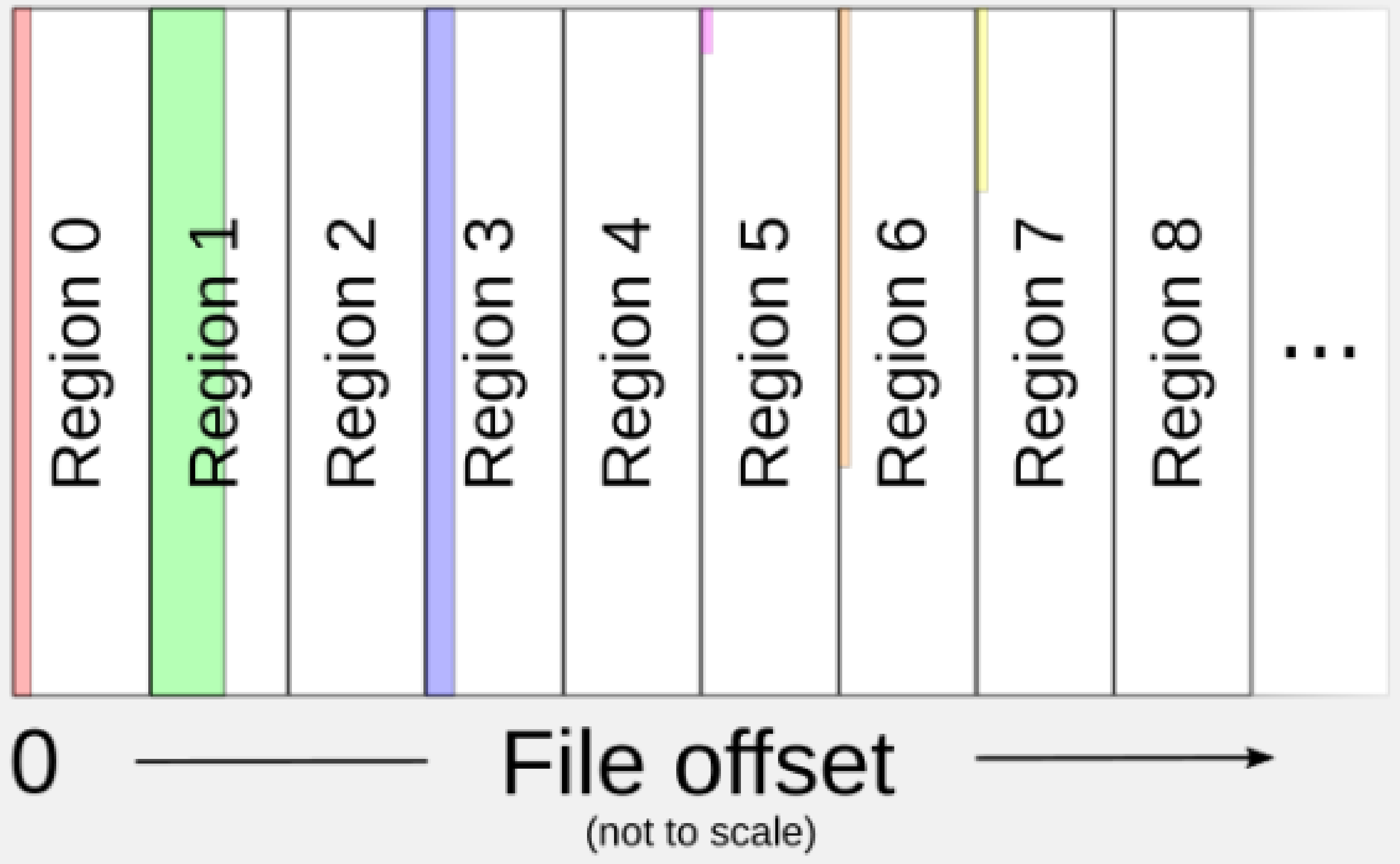
```
01:00.0 VGA compatible controller: NVIDIA Corporation GM107
 Region 0: Memory at f6000000 (32-bit, non-prefetchable) [size=16M]
 Region 1: Memory at e0000000 (64-bit, prefetchable) [size=256M]
 Region 3: Memory at f0000000 (64-bit, prefetchable) [size=32M]
 Region 5: I/O ports at e000 [size=128]
 Expansion ROM at f7000000 [disabled] [size=512K]
```

These are all regions

Even PCI config space itself is a region

Regions map to device file offsets

```
01:00.0 VGA compatible controller: NVIDIA Corporation GM107
 Region 0: Memory at f6000000 (32-bit, non-prefetchable) [size=16M]
 Region 1: Memory at e0000000 (64-bit, prefetchable) [size=256M]
 Region 3: Memory at f0000000 (64-bit, prefetchable) [size=32M]
 Region 5: I/O ports at e000 [size=128]
 Expansion ROM at f7000000 [disabled] [size=512K]
```



Properties discovered via ioctls

Properties discovered via ioctls

VFIO_DEVICE_GET_INFO

```
struct vfio_device_info
├── argsz
├── flags
│   ├── VFIO_DEVICE_FLAGS_PCI
│   ├── VFIO_DEVICE_FLAGS_PLATFORM
│   └── VFIO_DEVICE_FLAGS_RESET
├── num_irqs
└── num_regions
```

Properties discovered via ioctls

VFIO_DEVICE_GET_REGION_INFO

```
struct vfio_region_info
├── argsz
├── cap_offset
├── flags
│   ├── VFIO_REGION_INFO_FLAG_CAPS
│   ├── VFIO_REGION_INFO_FLAG_MMAP
│   ├── VFIO_REGION_INFO_FLAG_READ
│   └── VFIO_REGION_INFO_FLAG_WRITE
├── index
├── offset
└── size
```

Properties discovered via ioctls

VFIO_DEVICE_GET_IRQ_INFO

```
struct vfio_irq_info
├── argsz
├── count
├── flags
│   ├── VFIO_IRQ_INFO_AUTOMASKED
│   ├── VFIO_IRQ_INFO_EVENTFD
│   ├── VFIO_IRQ_INFO_MASKABLE
│   └── VFIO_IRQ_INFO_NORESIZE
└── index
```

Speaking of interrupts

Speaking of interrupts

Q: How does a device signal the driver?

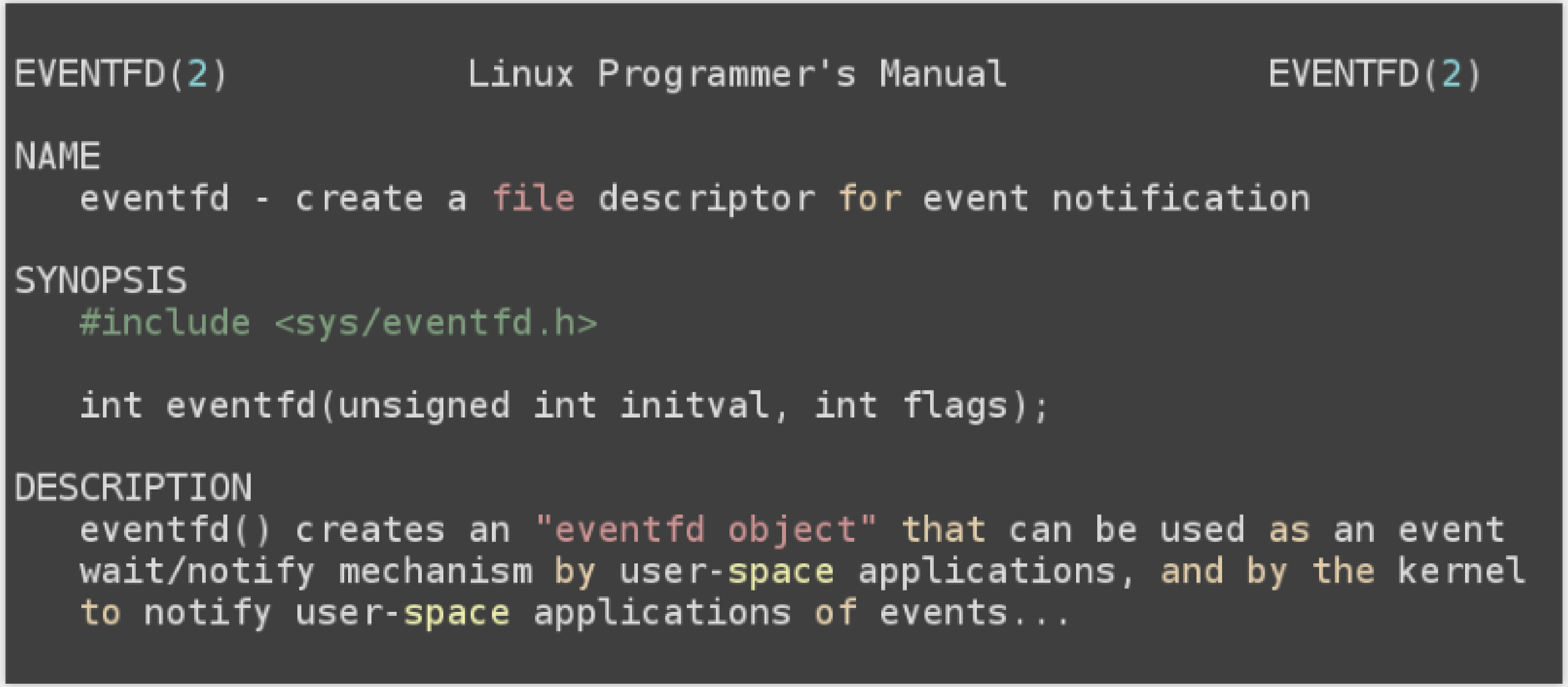
Speaking of interrupts

Q: How does a device signal the driver?

A: Interrupts

How do we interrupt userspace?

How do we interrupt userspace?



```
EVENTFD(2)            Linux Programmer's Manual            EVENTFD(2)

NAME
   eventfd - create a file descriptor for event notification

SYNOPSIS
   #include <sys/eventfd.h>

   int eventfd(unsigned int initval, int flags);

DESCRIPTION
   eventfd() creates an "eventfd object" that can be used as an event
   wait/notify mechanism by user-space applications, and by the kernel
   to notify user-space applications of events...
```

VFIO_DEVICE_SET_IRQS

```
struct vfio_irq_set
├── argsz
├── count
├── data[]
├── flags
│   ├── VFIO_IRQ_SET_ACTION_MASK
│   ├── VFIO_IRQ_SET_ACTION_TRIGGER
│   ├── VFIO_IRQ_SET_ACTION_UNMASK
│   ├── VFIO_IRQ_SET_DATA_BOOL
│   ├── VFIO_IRQ_SET_DATA_EVENTFD
│   └── VFIO_IRQ_SET_DATA_NONE
├── index
└── start
```

One remaining question

How does a device transfer data?

Direct Memory Access - DMA

- I/O device can read & write:
  - System memory (RAM)
  - Peer device memory
- Outside of CPU MMU control

Direct Memory Access - DMA

- I/O device can read & write:
  - System memory (RAM)
  - Peer device memory
- Outside of CPU MMU control

Need an MMU for I/O, an IOMMU

IOMMU Roles

IOMMU Roles

- Translation
  - I/O Virtual Address (IOVA) space
  - Previously the main purpose of an IOMMU

IOMMU Roles

- Translation
  - I/O Virtual Address (IOVA) space
  - Previously the main purpose of an IOMMU
- Isolation
  - Per device translation
  - Invalid accesses blocked

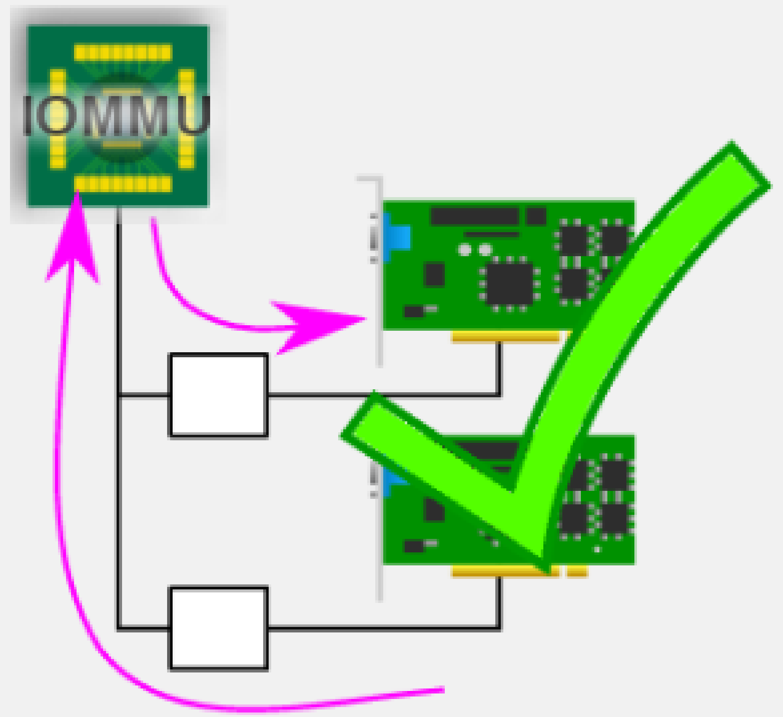
IOMMU Roles

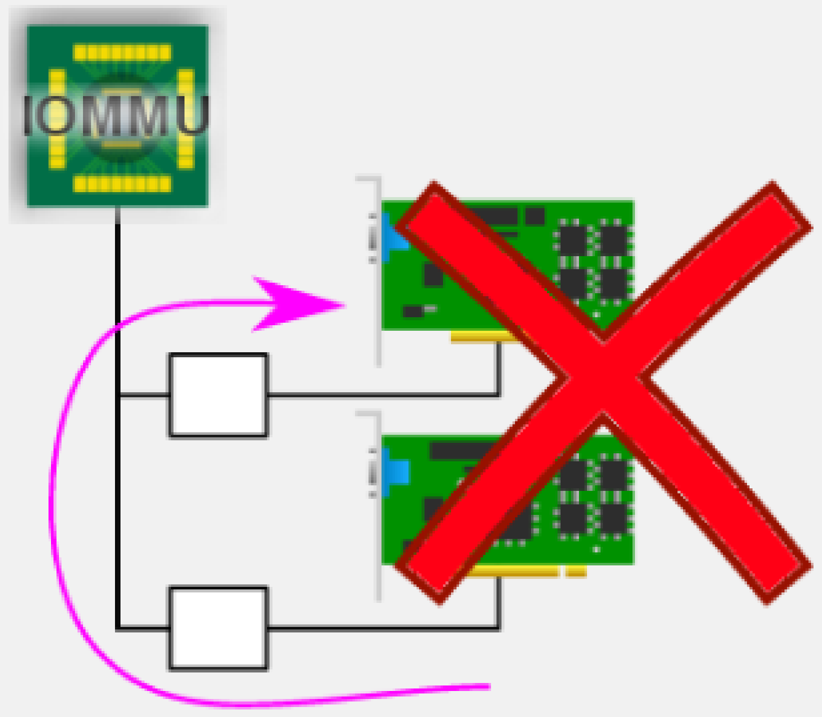
- Translation
  - I/O Virtual Address (IOVA) space
  - Previously the main purpose of an IOMMU
- Isolation
  - Per device translation
  - Invalid accesses blocked

Both required for secure user access

IOMMU Issues

- DMA Aliasing
  - Not all devices generate unique IDs
  - Not all devices generate the ID they should
- DMA Isolation
  - Peer-to-peer DMA translation





Solution: IOMMU groups

- Group of devices with DMA isolation from other groups
- Grouping determined by IOMMU driver
  - Not user configurable
- Influencing factors:
  - IOMMU capabilities
  - Endpoint device isolation
  - Bus and interconnect properties
- Heavily influences VFIO design

Memory Issues

- IOVA page faults are not supported end-to-end
    - IOVA to physical mappings are static
- User memory can be relocated
    - Swapping, page merging, etc

Memory Issues

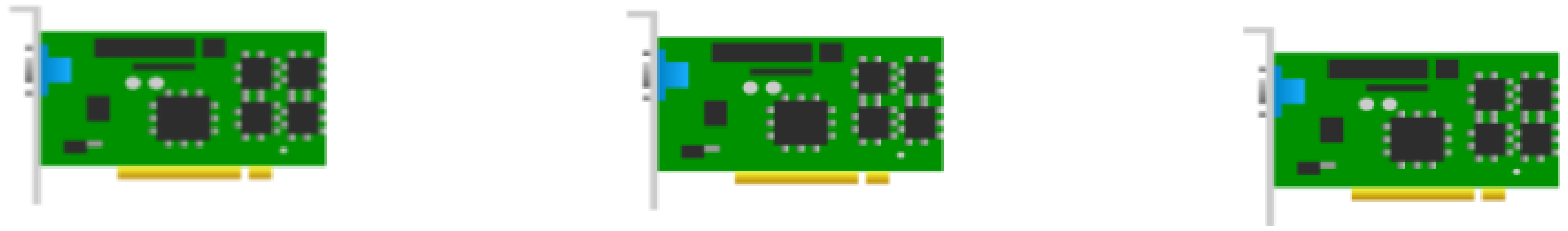
- IOVA page faults are not supported end-to-end
  - IOVA to physical mappings are static
- User memory can be relocated
  - Swapping, page merging, etc

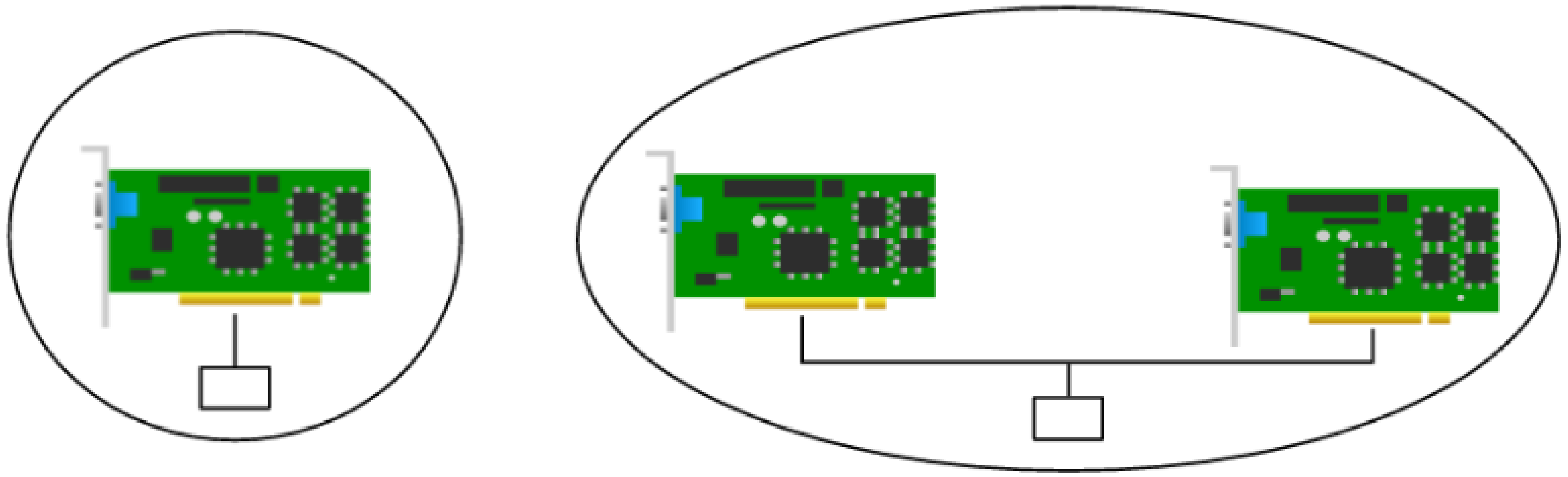
Solution: Page pinning

A few downsides

- Pinned memory is locked memory
  - User requires sufficient locked memory limits
- Prevents page merging and swapping
  - As intended, but we like those features

Let's walk through an example

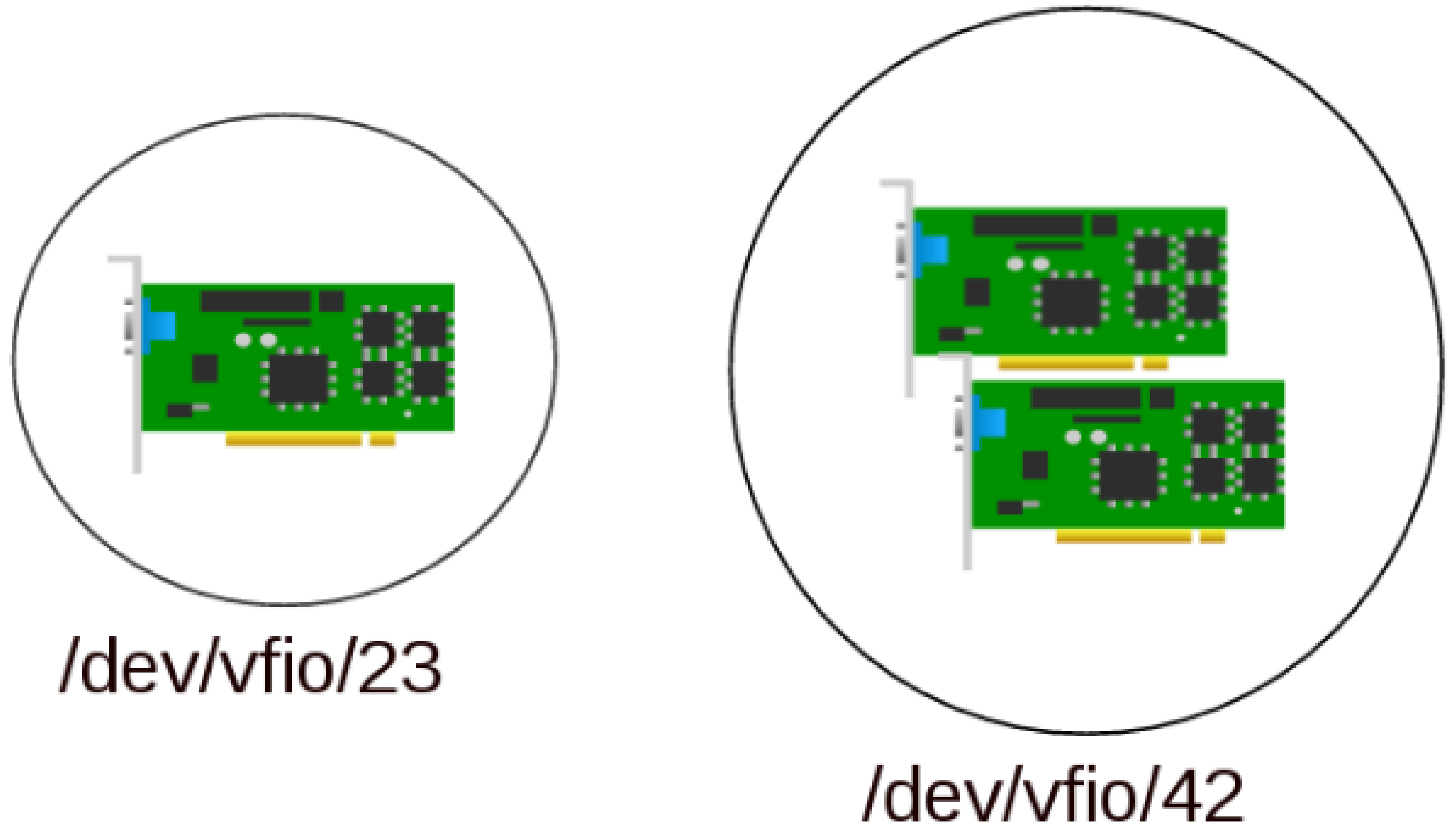


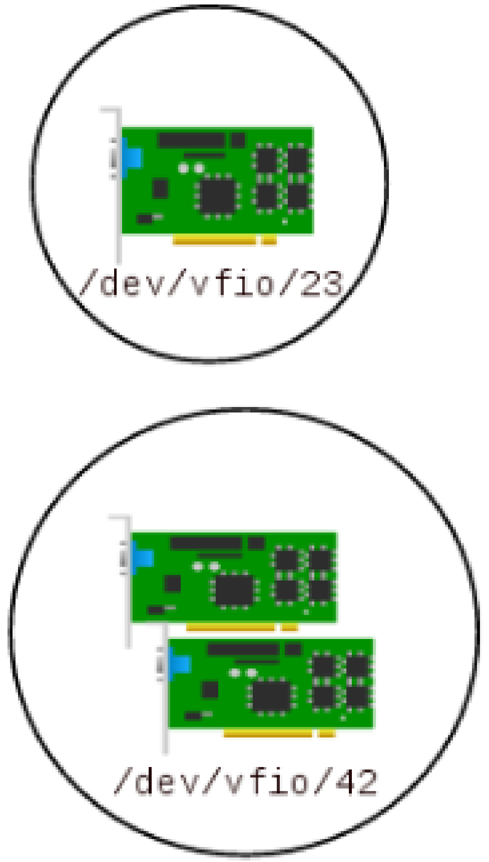


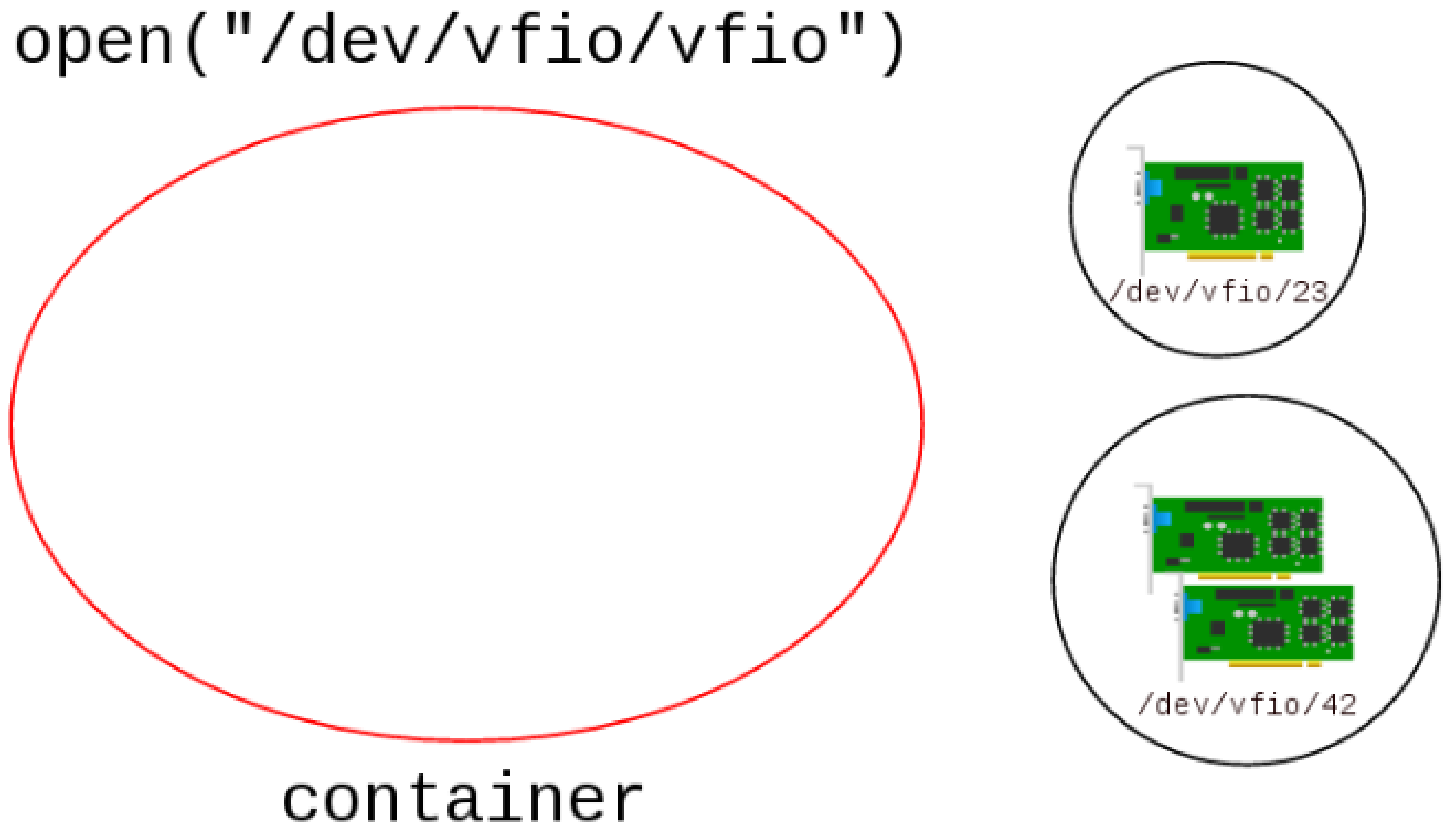
IOMMU grouping considerations:

- IOMMU visibility
- DMA isolation

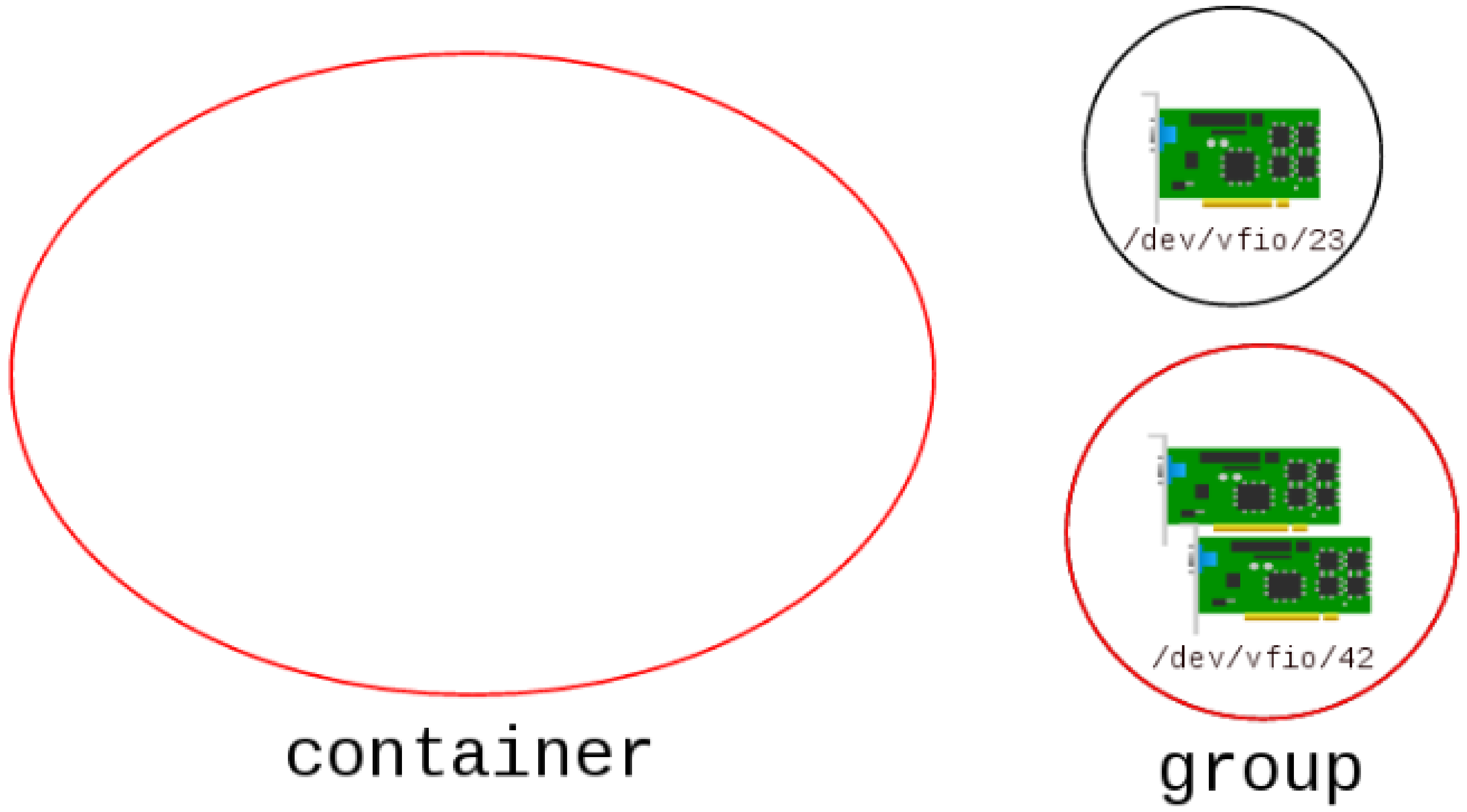
Binding devices to vfio-pci results in vfio group nodes



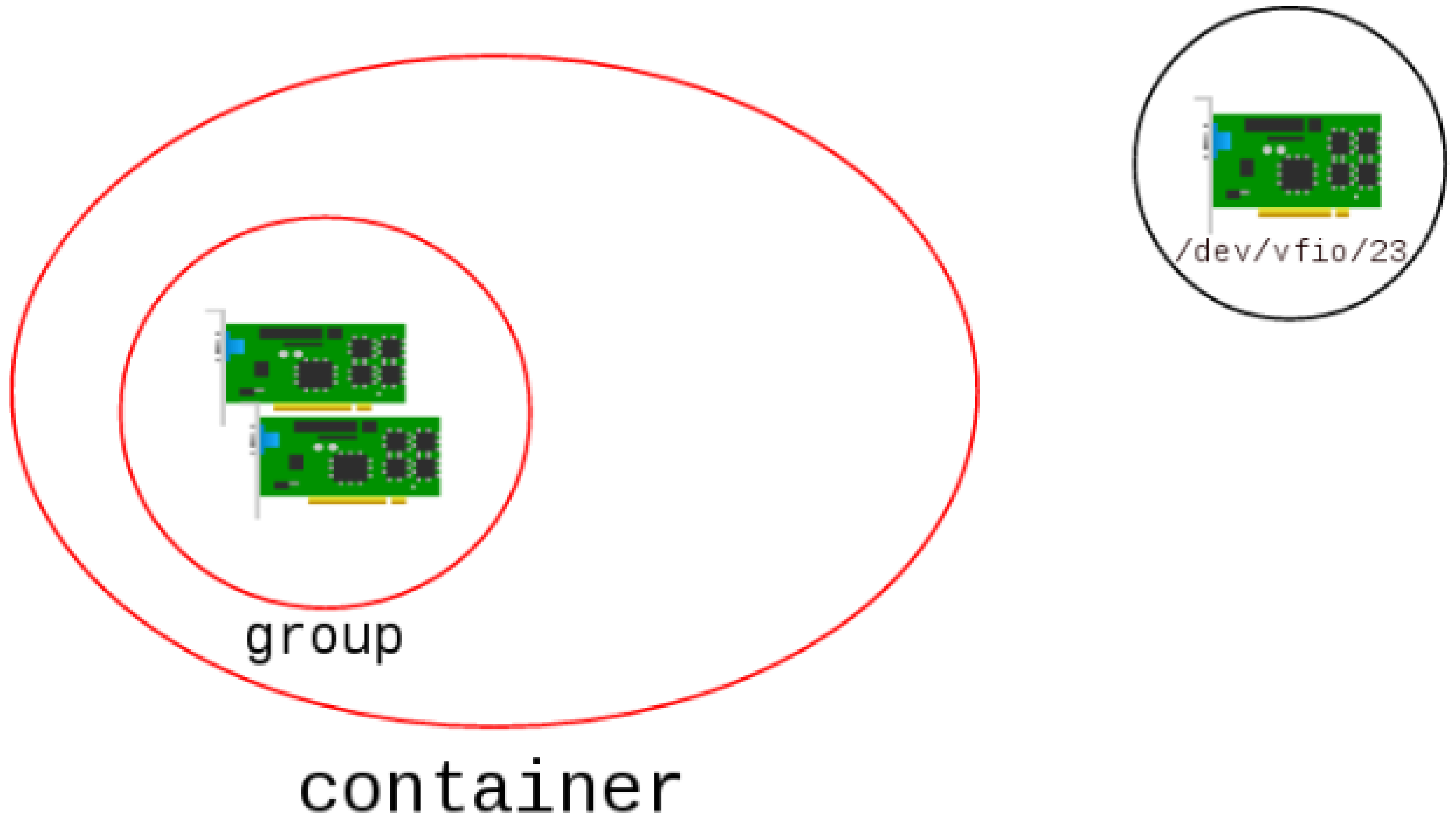

/dev/vfio/23
/dev/vfio/42


open("/dev/vfio/vfio")
/dev/vfio/23
/dev/vfio/42
container

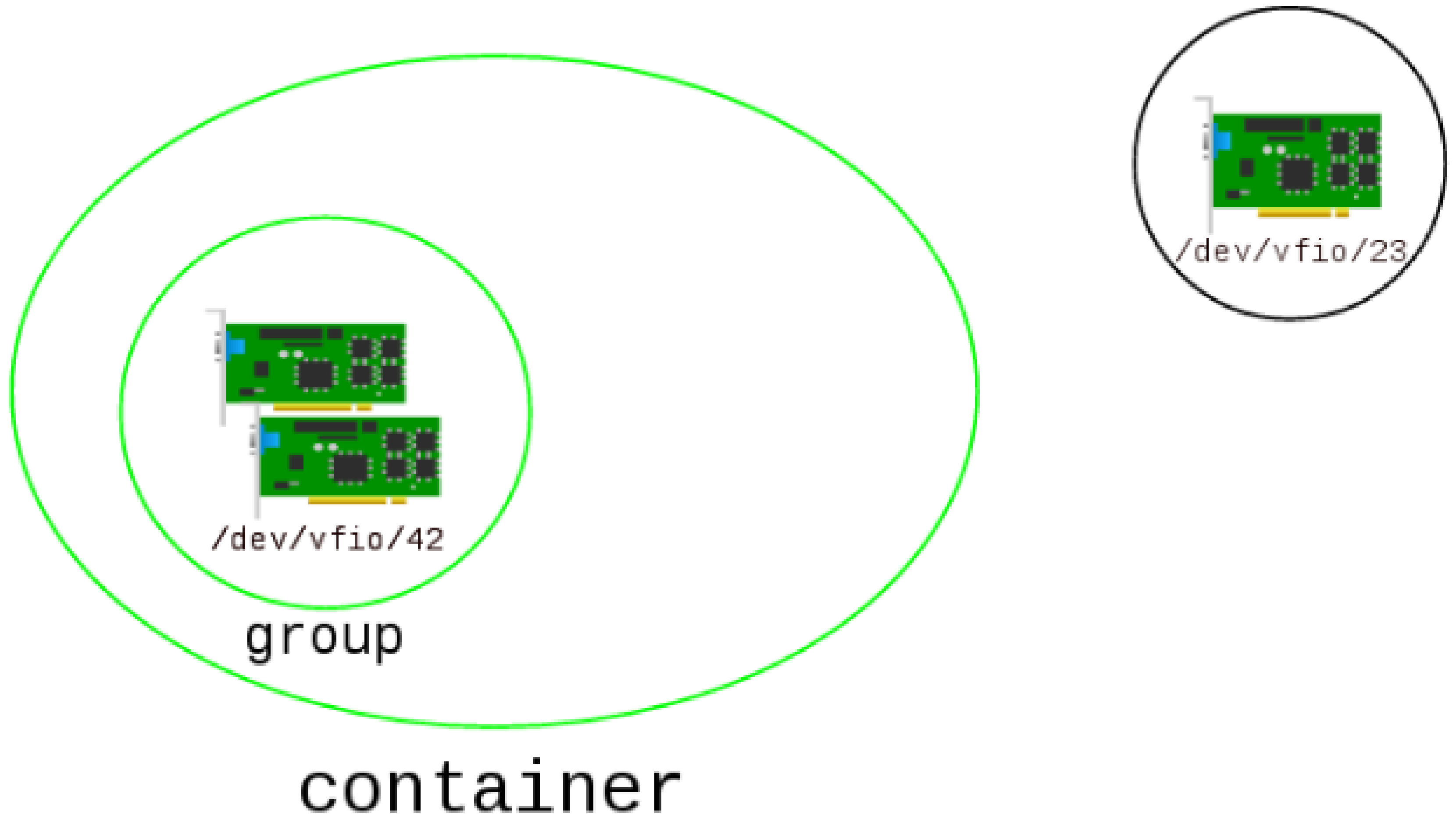
open("/dev/vfio/42")

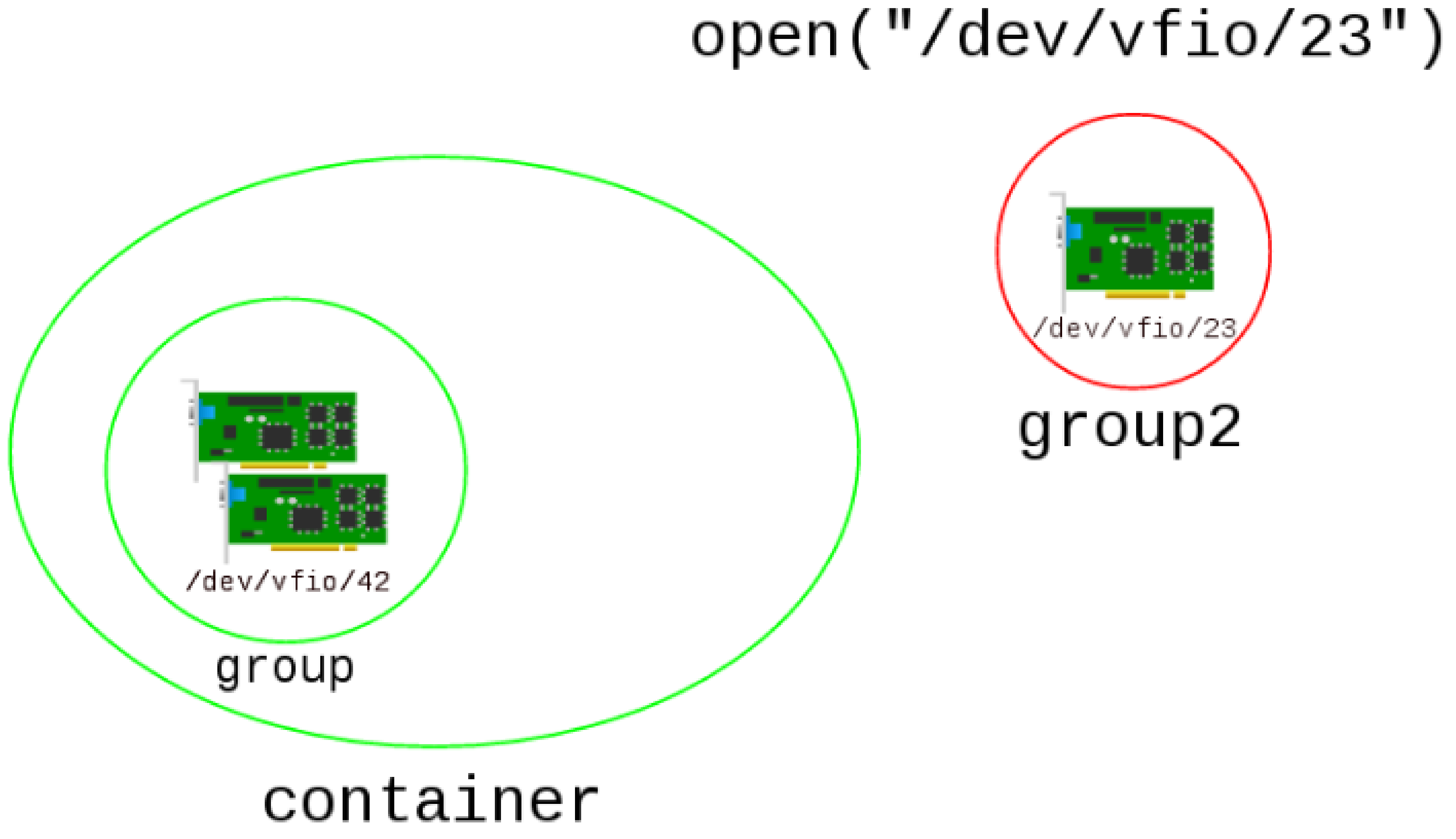


ioctl(group,VFIO_GROUP_SET_CONTAINER,&container)

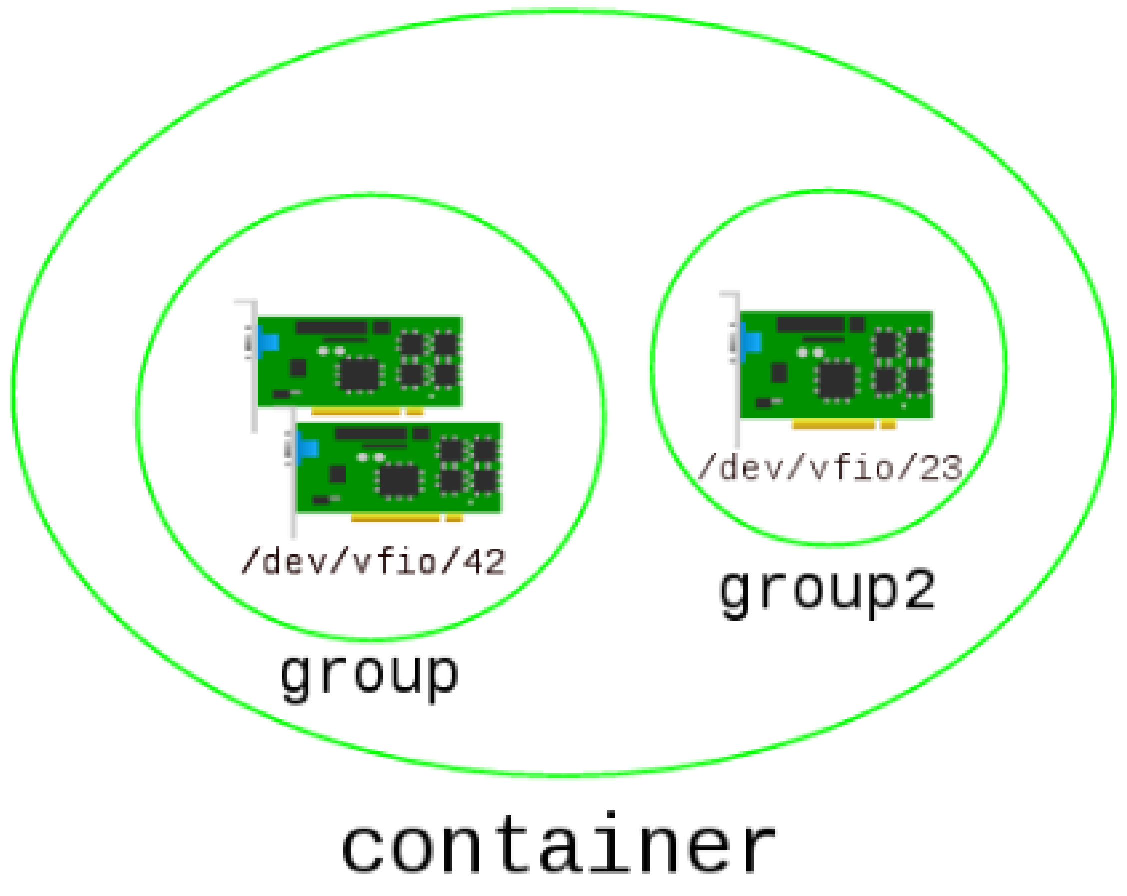


```
ioctl(container,VFIO_SET_IOMMU,VFIO_TYPE1_IOMMU)
```

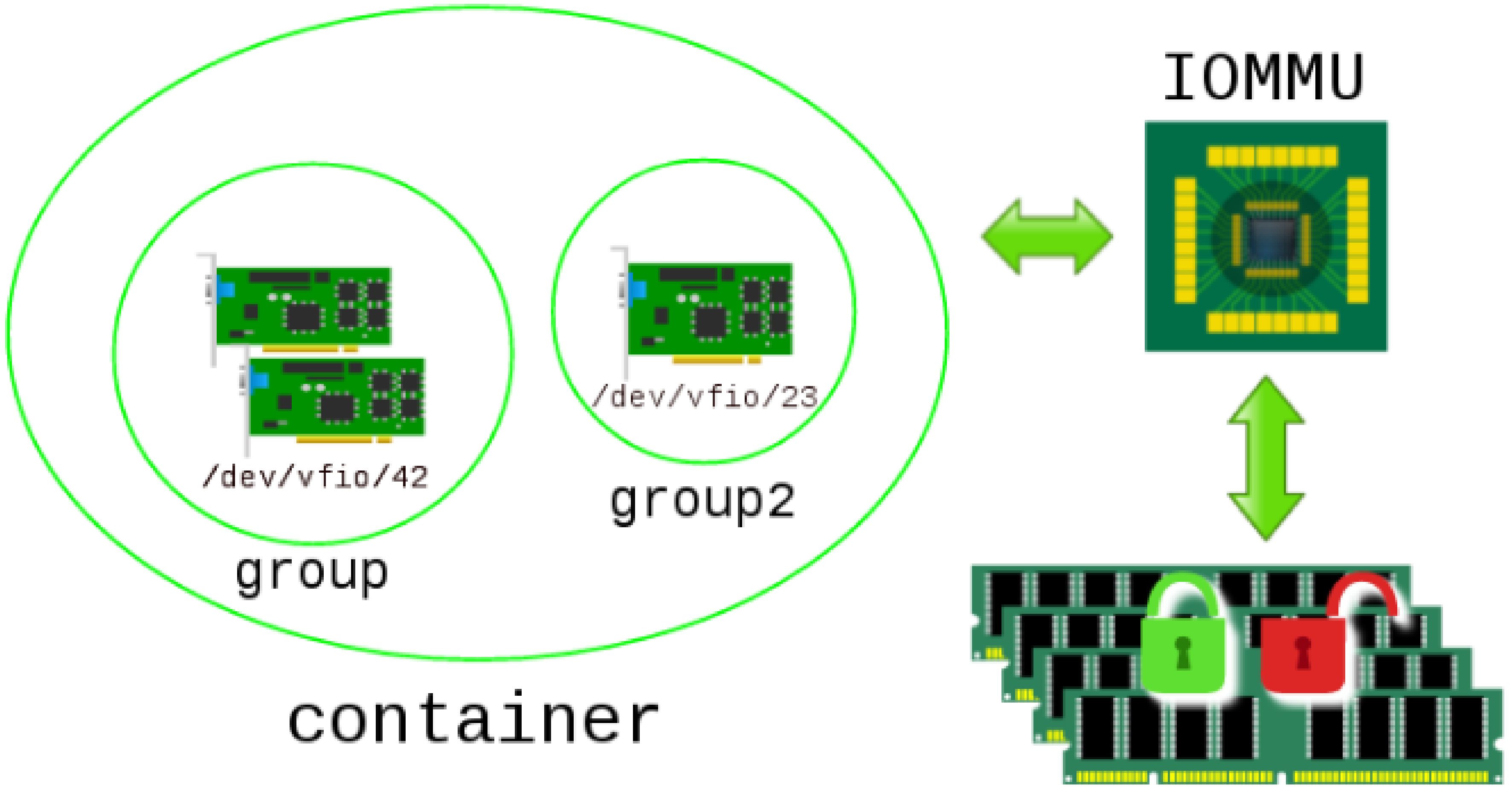



open("/dev/vfio/23")
/dev/vfio/42
group
container
/dev/vfio/23
group2

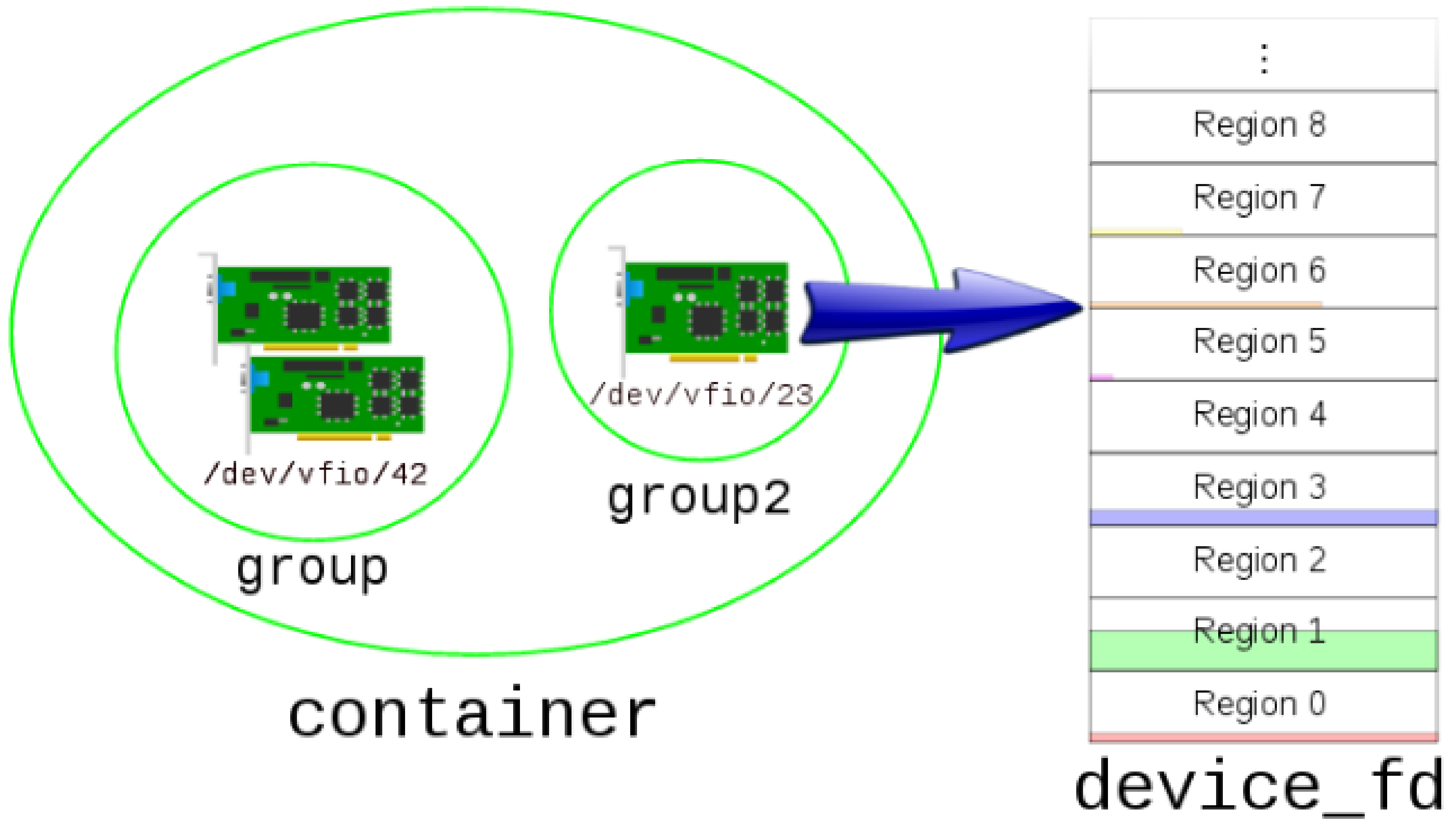
```
ioctl(group2,VFIO_GROUP_SET_CONTAINER,&container)
```



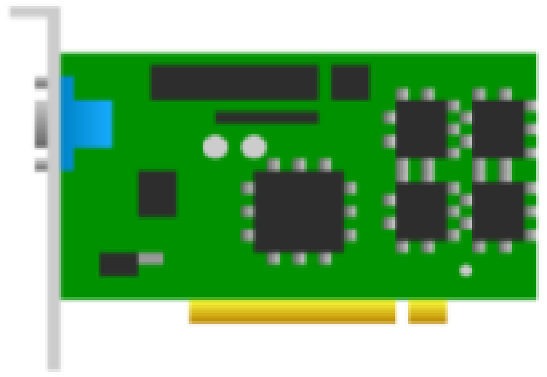
```
ioctl(container,VFIO_IOMMU_MAP_DMA,&map)
ioctl(container,VFIO_IOMMU_UNMAP_DMA,&unmap)
```



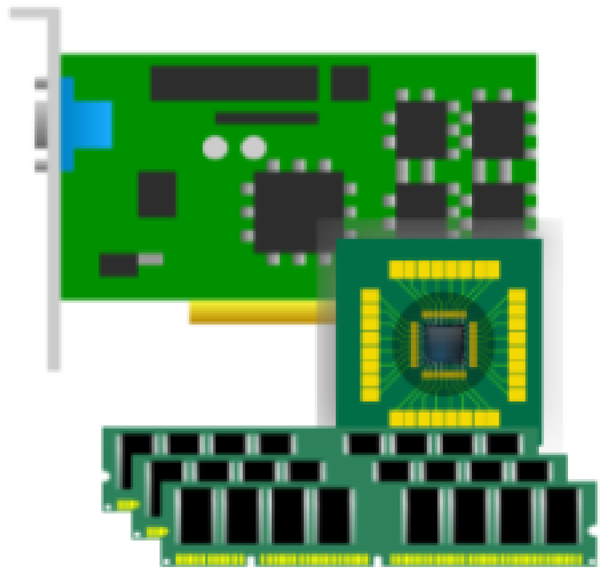
```
ioctl(group2,
          VFIO_GROUP_GET_DEVICE_FD,"0000:01:00.0")
```



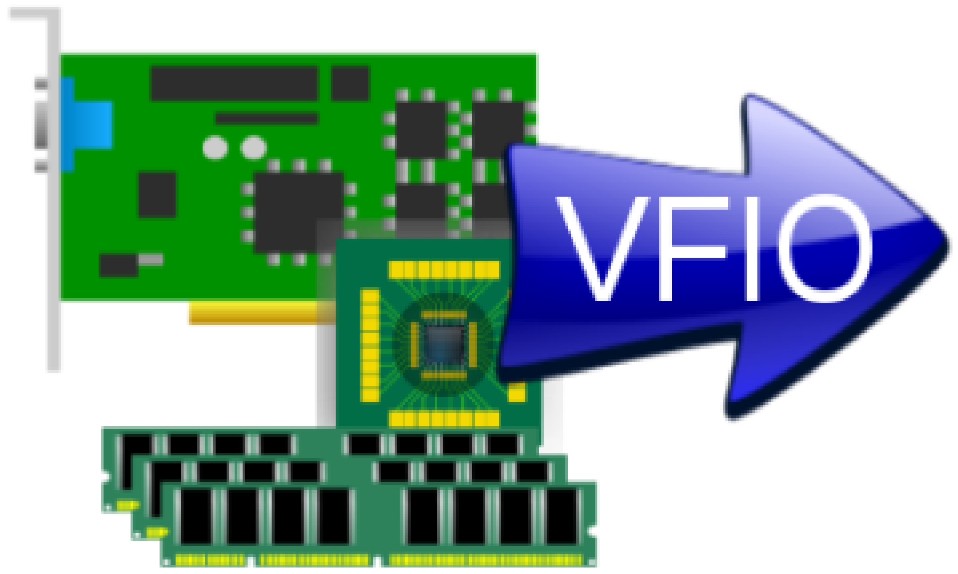
VFIO *in a nutshell*



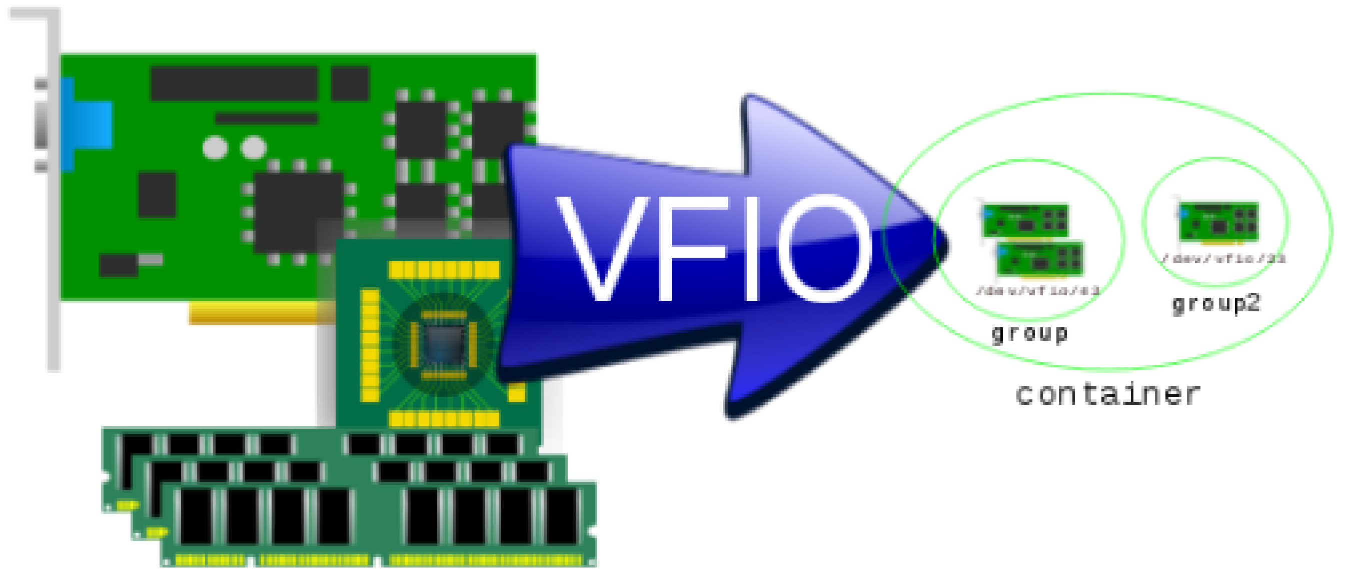
VFIO *in a nutshell*



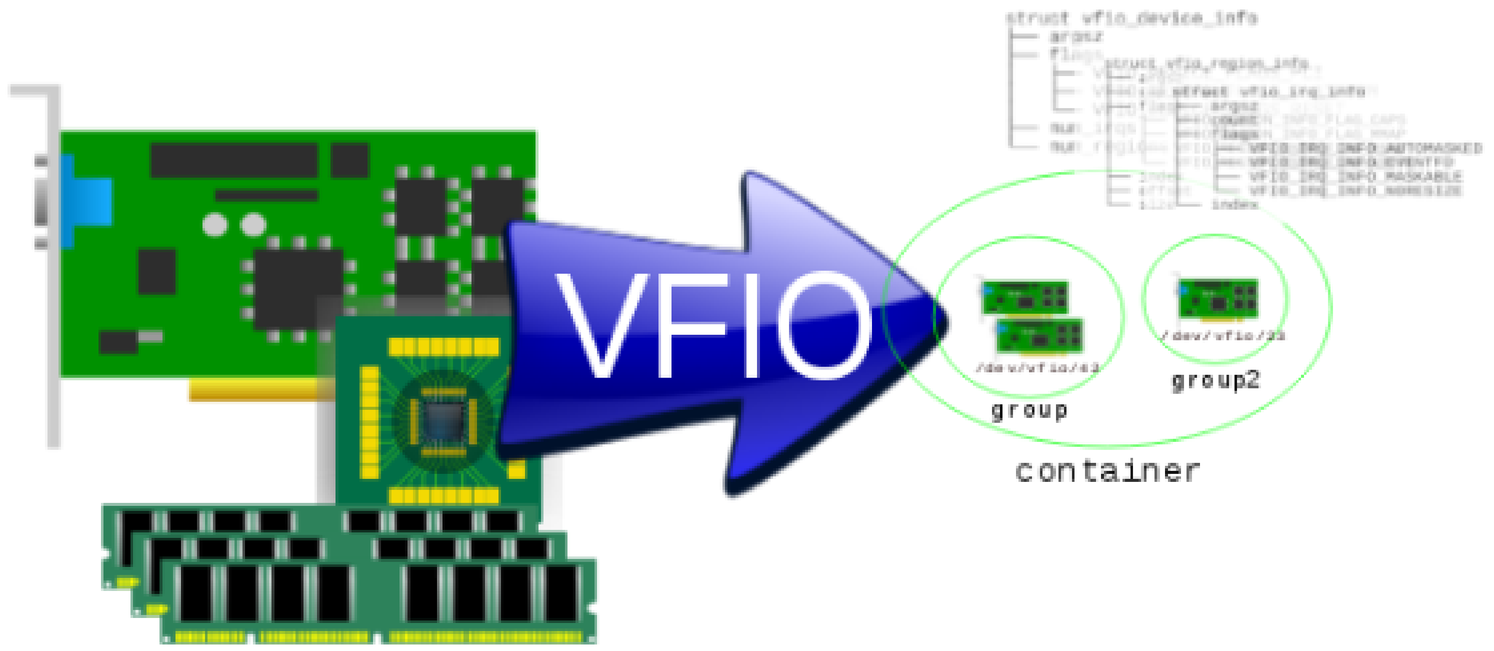
VFIO *in a nutshell*



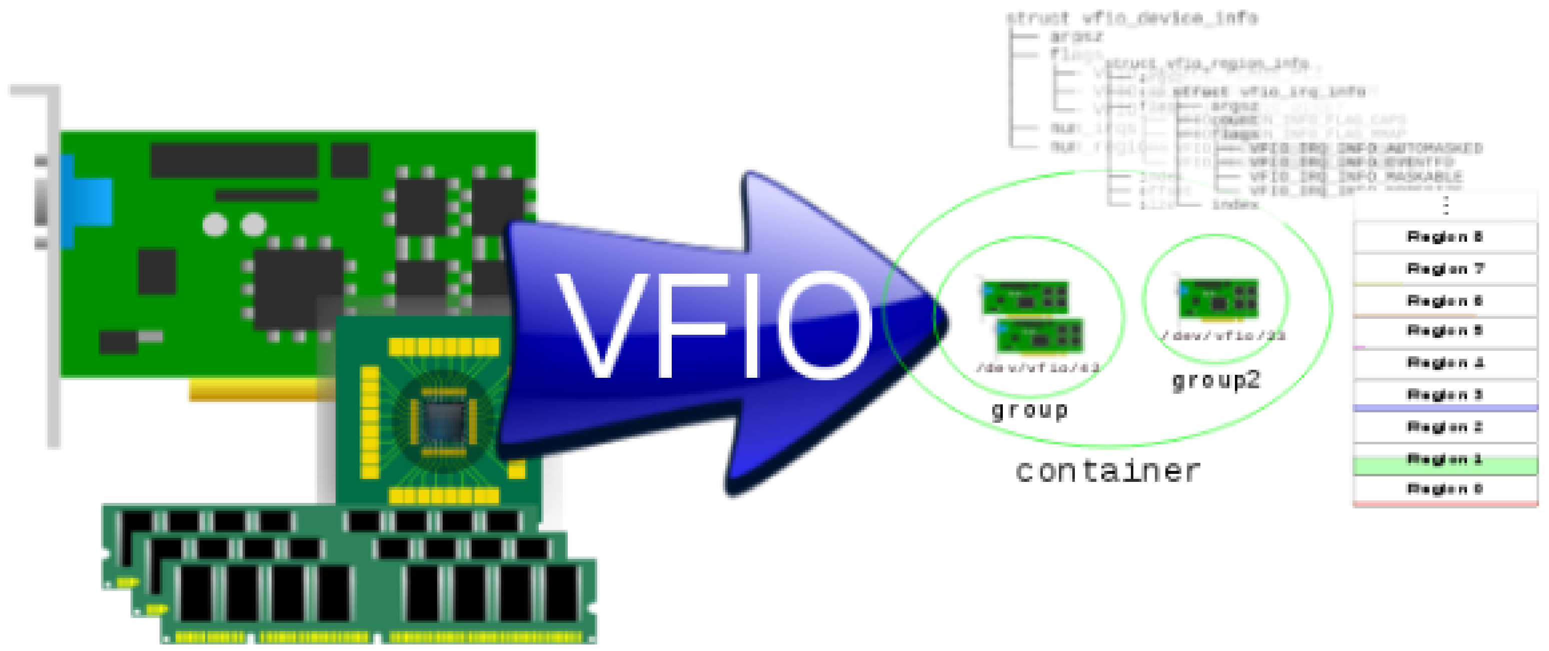
VFIO *in a nutshell*



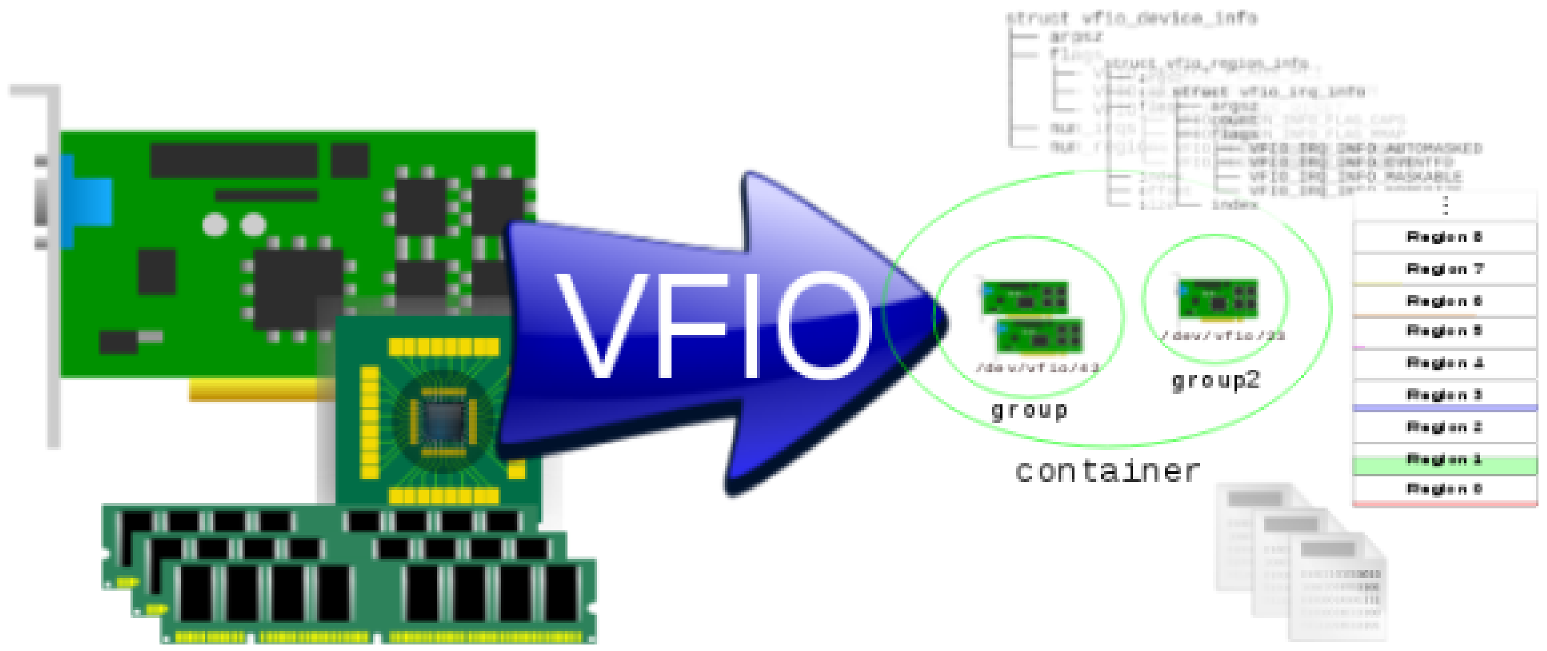
VFIO *in a nutshell*



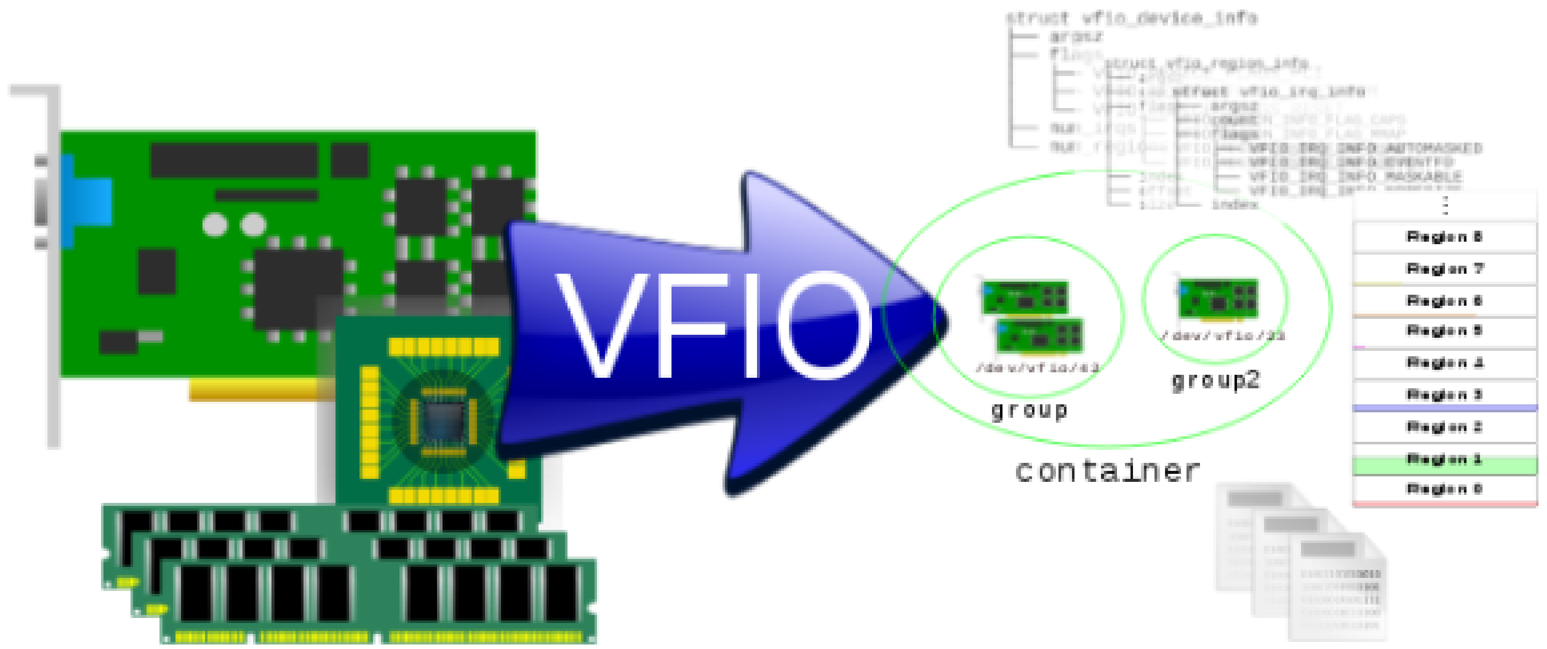
VFIO *in a nutshell*



VFIO *in a nutshell*



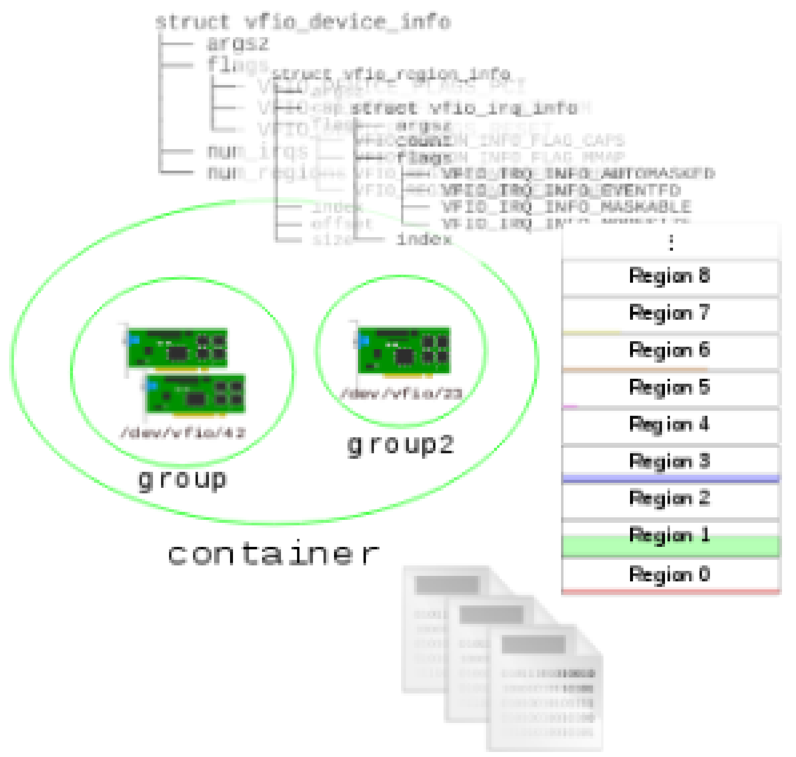
VFIO *in a nutshell*



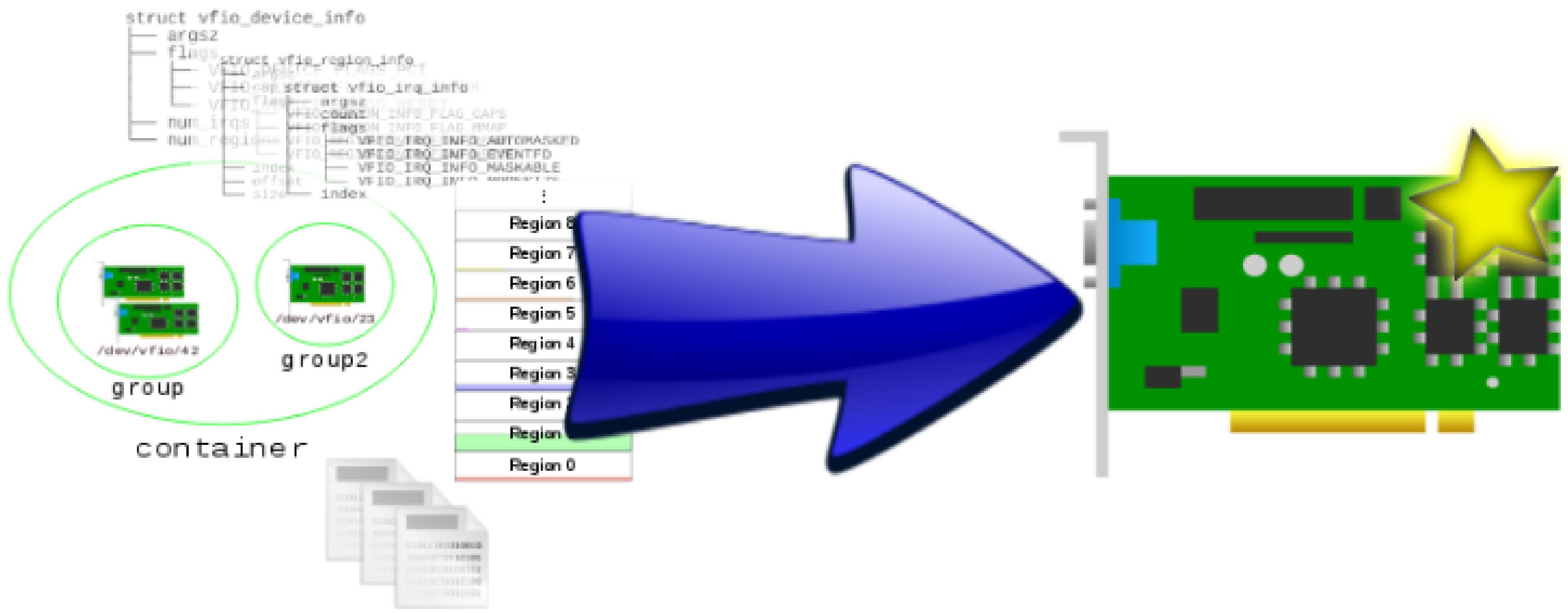
A device decomposed

Recomposing a device

Deconstructed device in userspace...



Deconstructed device in userspace...



...turns into assigned device

QEMU

Quick EMUlator

Quick EMUlator

"Creating fake devices since 2003"

Same questions, different perspective

- How does the guest program a device?
- How does a device signal the guest?
- How does a device transfer data?

Device Programming

How does VM programmed I/O reach a device?

Device Programming

How does VM programmed I/O reach a device?

- Trapped by hypervisor (QEMU/KVM)

Device Programming

How does VM programmed I/O reach a device?

- Trapped by hypervisor (QEMU/KVM)
- MemoryRegion lookup performed

Device Programming

How does VM programmed I/O reach a device?

- Trapped by hypervisor (QEMU/KVM)
- MemoryRegion lookup performed
- MemoryRegion.{read,write} accessors called

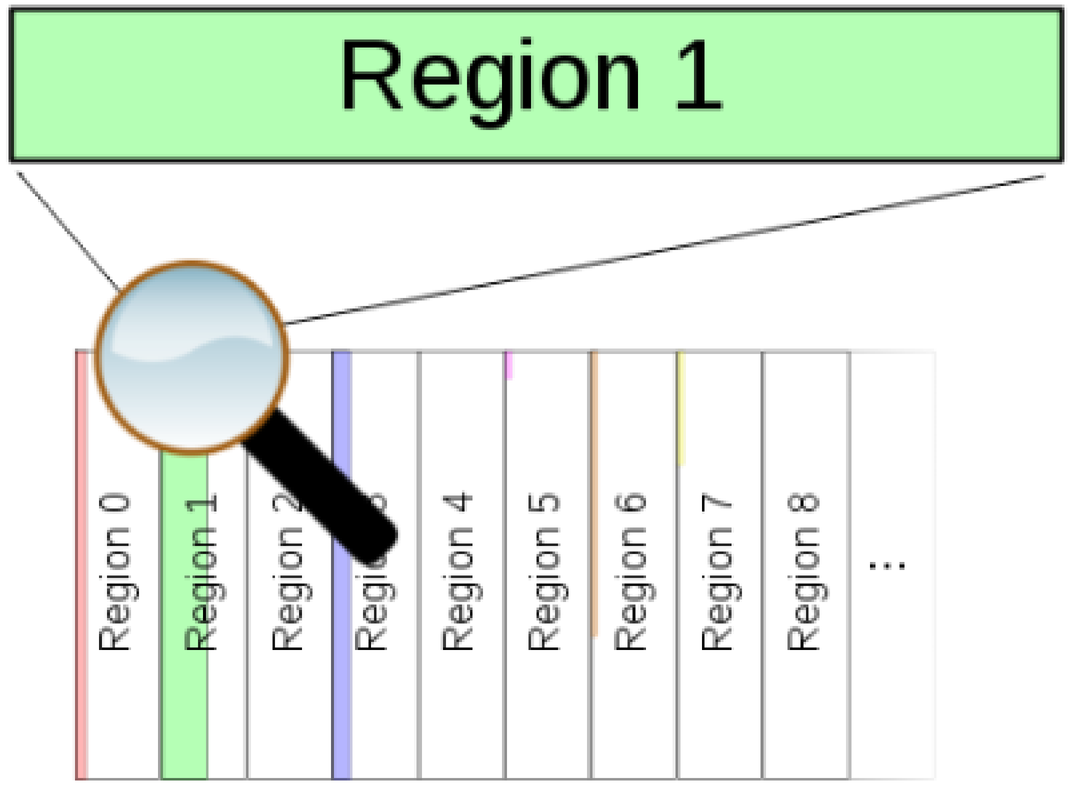
Device Programming

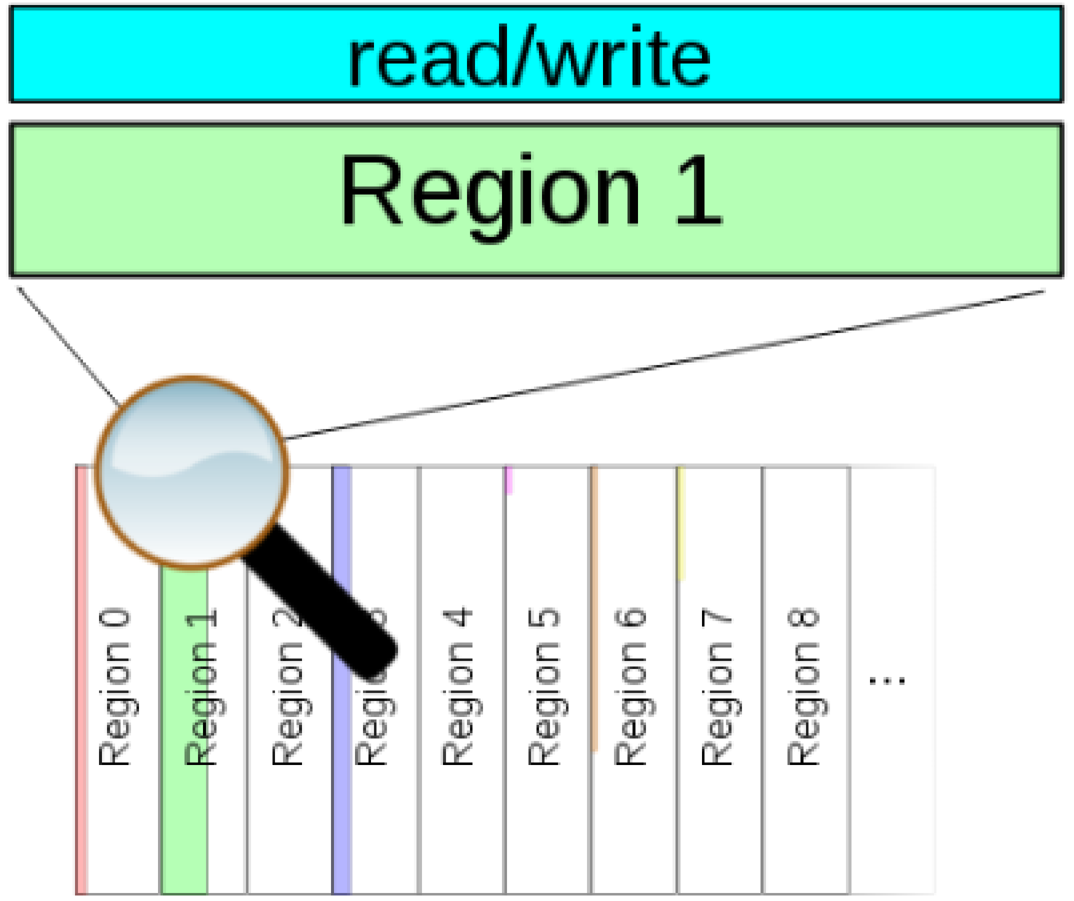
How does VM programmed I/O reach a device?

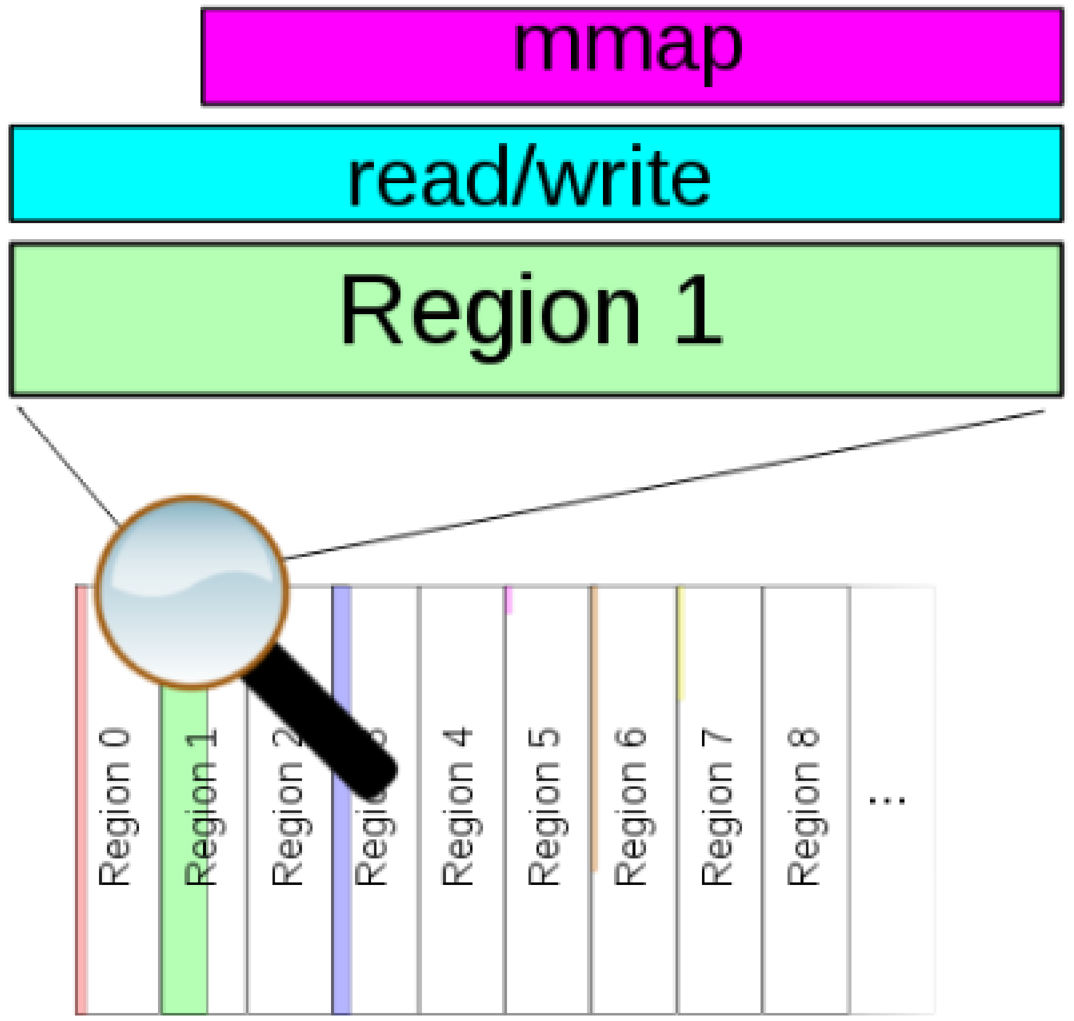
- Trapped by hypervisor (QEMU/KVM)
- MemoryRegion lookup performed
- MemoryRegion.{read,write} accessors called
- read/write to vfio region offsets

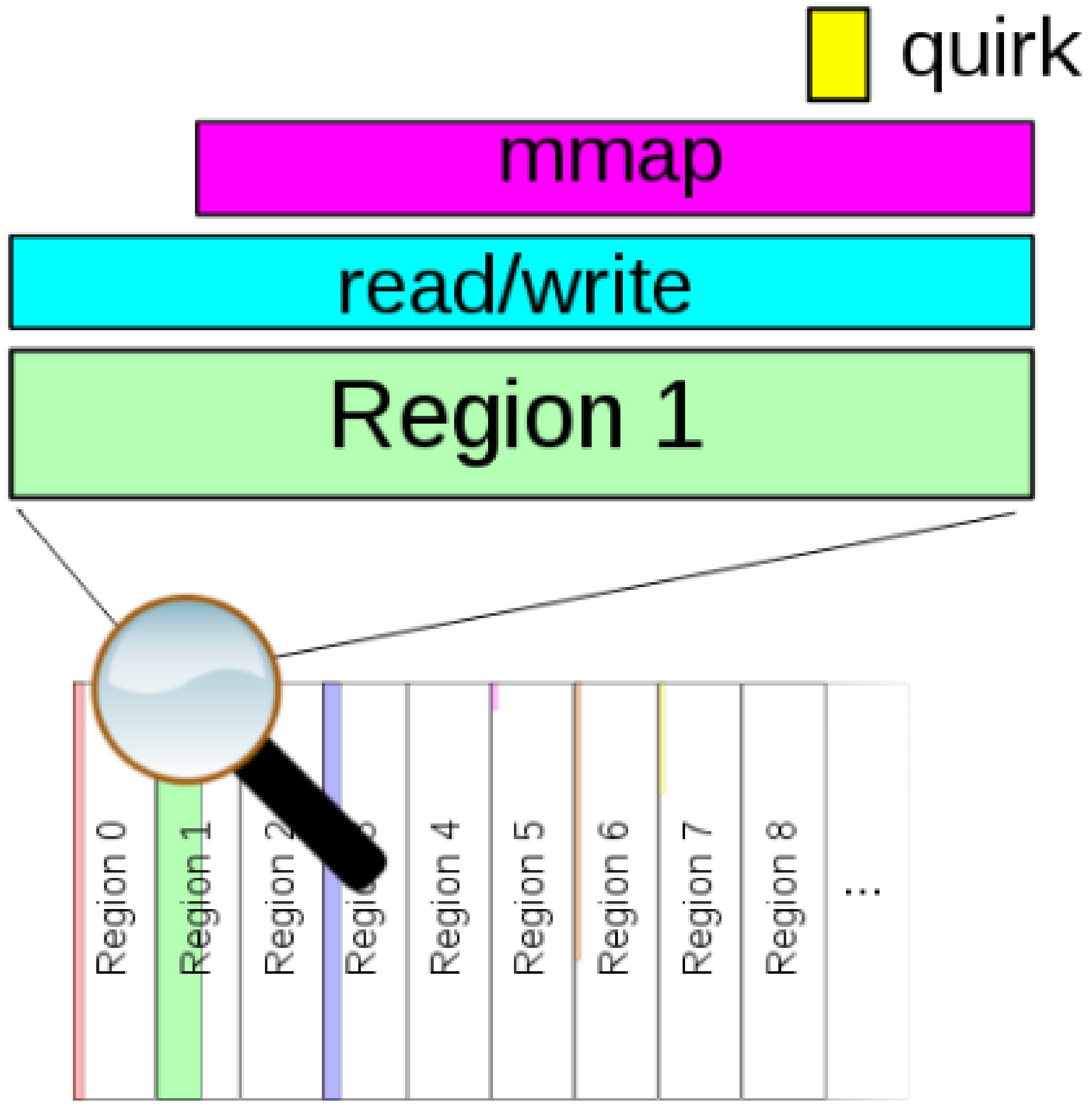
MemoryRegion Layering

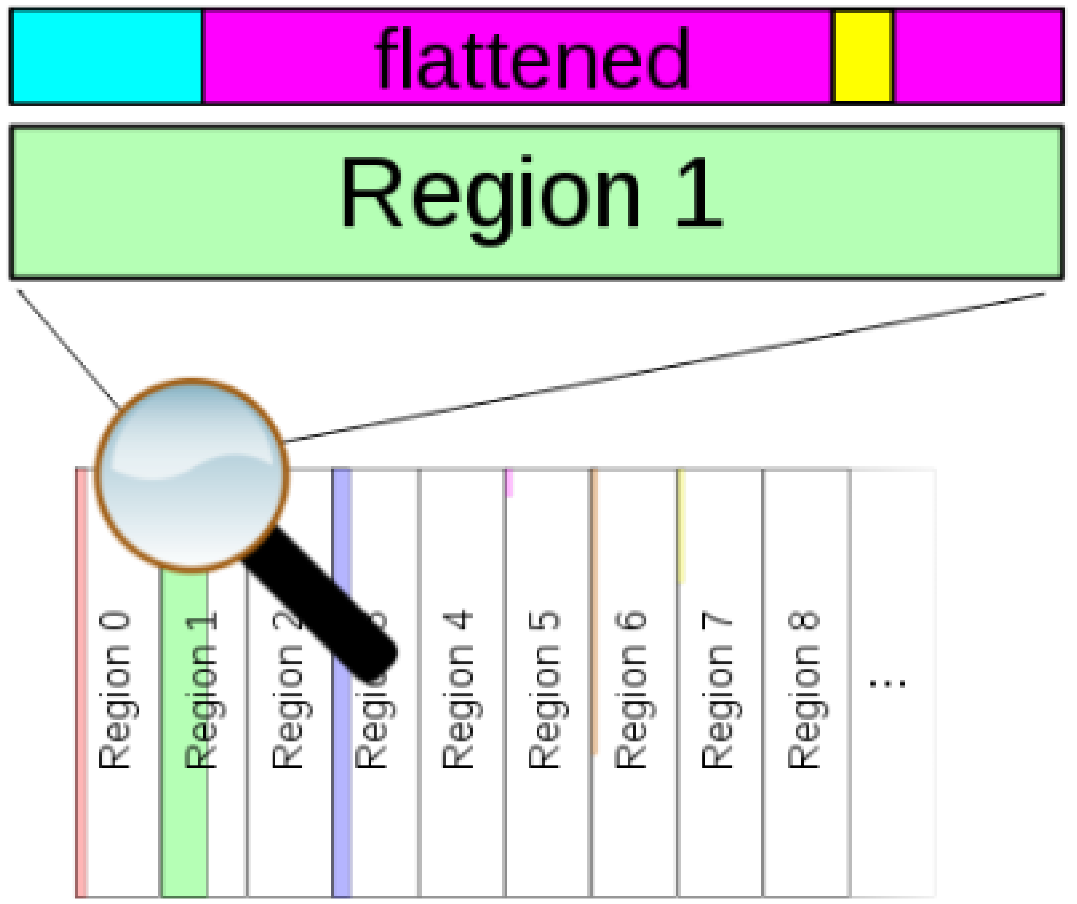
- "Slow" read/write base layer
- "Fast" mmap overlay
- "Quirks" to correct device virtualization issues

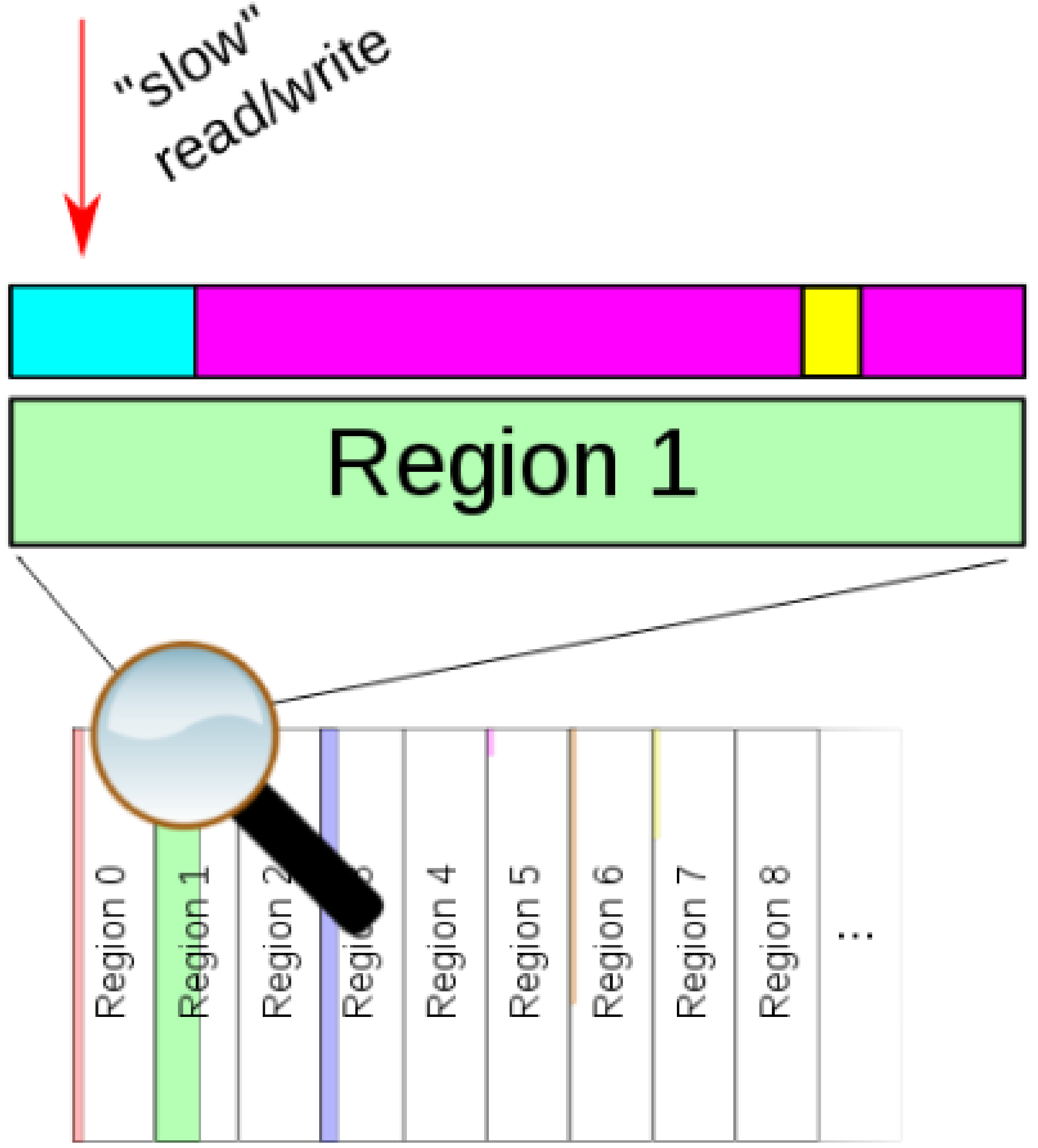

Region 1
Region 0
Region 1
Region 2
Region 3
Region 4
Region 5
Region 6
Region 7
Region 8
...

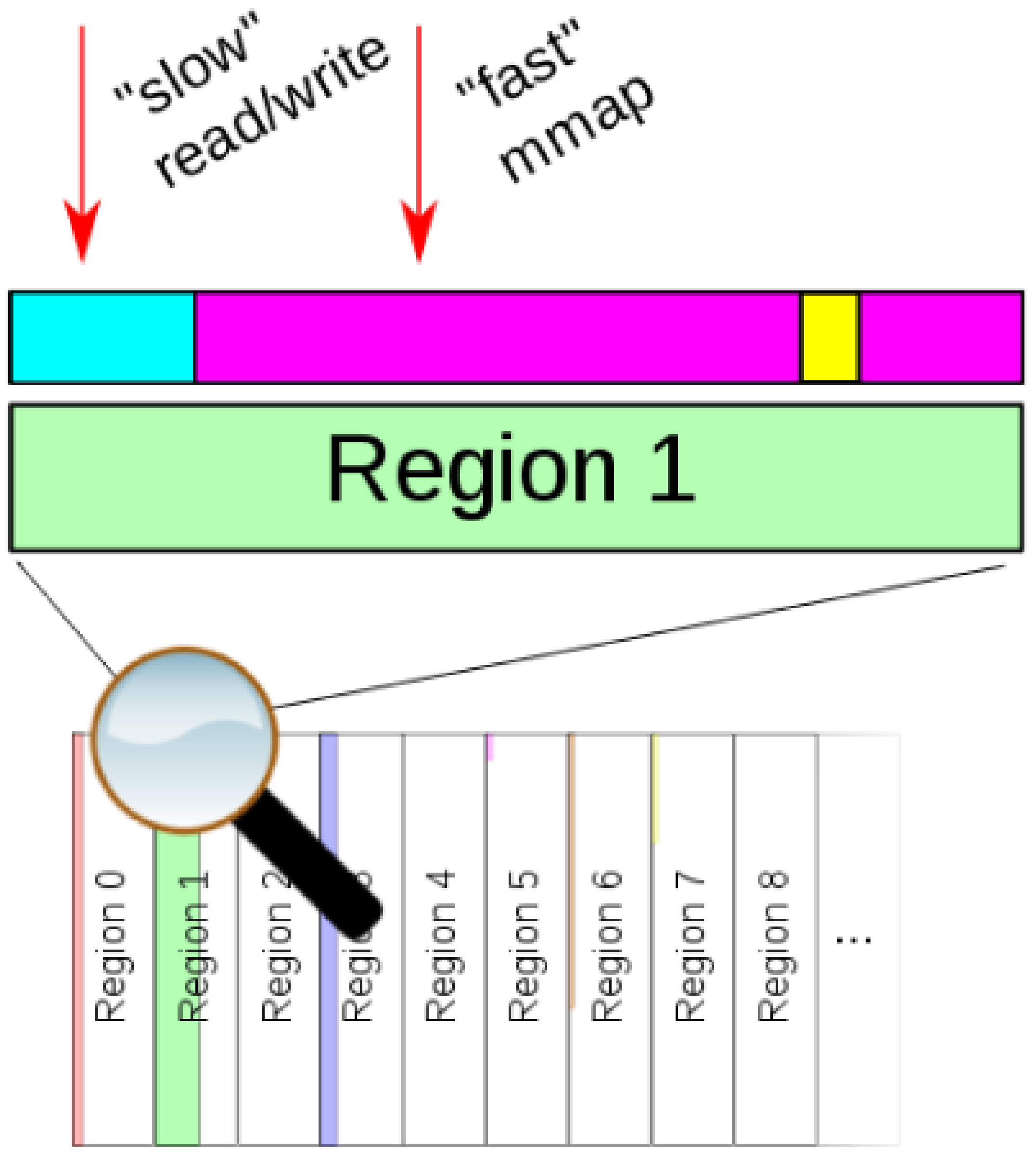

read/write
Region 1
Region 0
Region 1
Region 2
Region 3
Region 4
Region 5
Region 6
Region 7
Region 8
...

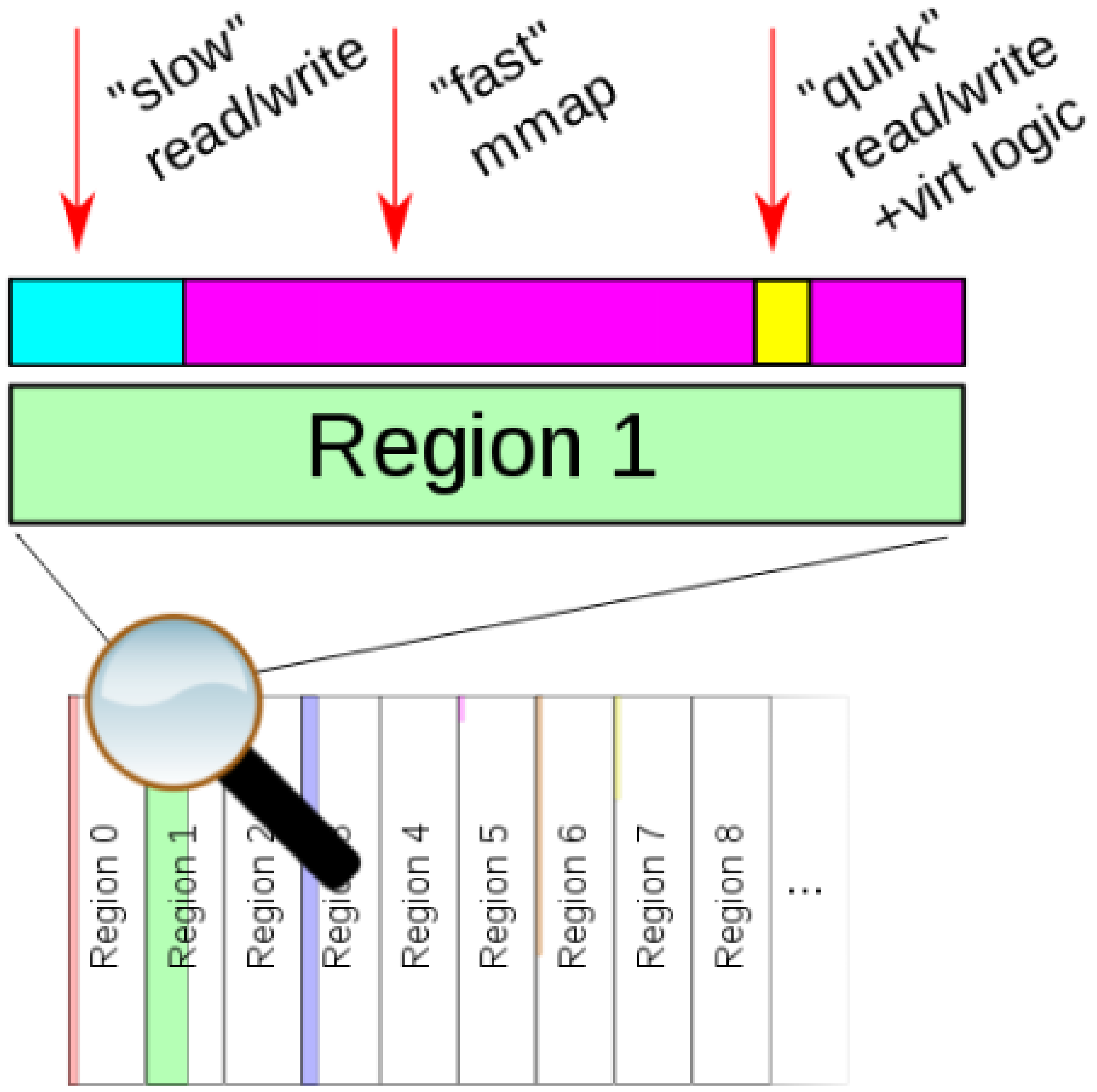

mmap
read/write
Region 1
Region 0
Region 1
Region 2
Region 3
Region 4
Region 5
Region 6
Region 7
Region 8
...


quirk
mmap
read/write
Region 1
Region 0
Region 1
Region 2
Region 3
Region 4
Region 5
Region 6
Region 7
Region 8
...


flattened
Region 1
Region 0
Region 1
Region 2
Region 3
Region 4
Region 5
Region 6
Region 7
Region 8
...


"slow"
read/write
Region 1
Region 0
Region 1
Region 2
Regi
Region 4
Region 5
Region 6
Region 7
Region 8
...


"slow"
read/write
"fast"
mmap
Region 1
Region 0
Region 1
Region 2
Regi
Region 4
Region 5
Region 6
Region 7
Region 8
...


"slow" read/write
"fast" mmap
"quirk" read/write +virt logic
Region 1
Region 0
Region 1
Region 2
Region 3
Region 4
Region 5
Region 6
Region 7
Region 8
...

Device Programming

How do guest PCI Config accesses reach a device?

- Not handled as MemoryRegions (yet)
- Selective handling
  - Direct pass-through
    - read/write to config region
  - Emulation & Virtualization
    - MSI/X, BARs, ROM, etc.

Interrupt Signaling

- QEMU configures vfio interrupt ioctl for device state
- Interrupts signal via eventfd
- EventNotifiers trigger QEMU device interrupts
- Two step process
  - host → QEMU, QEMU → VM
- How to make it faster?

irqfd

- eventfds signal events
- irqfds receive event signals
- eventfds can signal irqfds
- KVM supports VM interrupts through irqfd
- One step process: host → KVM
- No exit to userspace

Accelerating IRQs in hardware

Accelerating IRQs in hardware

- APIC Virtualization (Intel APICv)
  - Exit-less interrupts into VM

Accelerating IRQs in hardware

- APIC Virtualization (Intel APICv)
  - Exit-less interrupts into VM
- VT-d Posted Interrupts
  - Interrupts direct to vCPU

Last question

How does a device transfer data?

Enabling DMA for the VM

Transparent VM mapping

- Map entire guest physical address space
- No IOMMU visible to guest
  - DMA is guest physical
  - Host IOMMU maps guest physical to host physical
- Accomplished through QEMU MemoryListeners

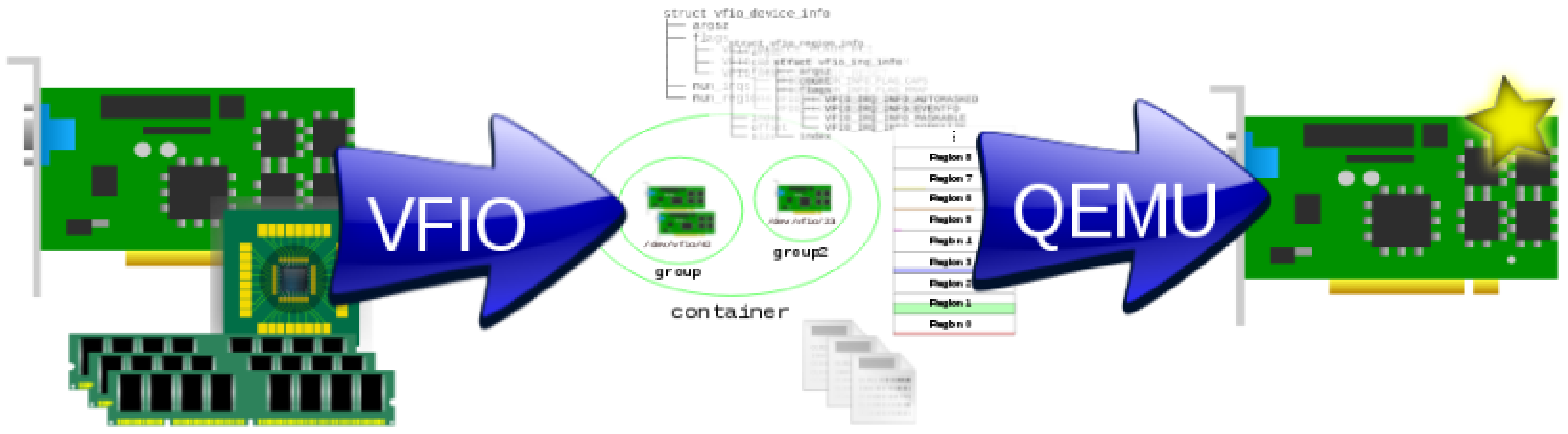
Summary

- How does the guest program a device?
  QEMU: MemoryRegions*
  VFIO: Device file descriptor regions
- How does a device signal the guest?
  QEMU: EventNotifiers
  VFIO/KVM: Eventfds/irqfds configured via ioctls
- How does a device transfer data?
  QEMU: MemoryListeners
  VFIO: IOMMU mapping & pinning ioctls

*Except PCI configuration space

PCI Device assignment with VFIO

The Complete Picture



kernel | userspace

What's new for 2016?

Intel Graphics Device (IGD) Assignment

- Sandy Bridge+: "Legacy" mode
  - Host: Linux v4.6+, QEMU 2.7+
- Broadwell+: Universal Passthrough (UPT) mode
- See http://vfio.blogspot.com for details

No new *Code 43* problems!

Workaround for NVIDIA Hyper-V detection "bug"

```
-cpu ...,hv_vendor_id=KeenlyKVM
```

```
<hyperv>
  ...
  <vendor_id state='on' value='KeenlyKVM'/>
  ...
</hyperv>
```

QEMU 2.5+, libvirt v1.3.3+

"Mediated Device" Development

vGPU on KVM - A VFIO Based Framework
by Neo Jia & Kirti Wankhede from NVIDIA
Thursday 10am

- Collaboration with Intel, IBM, and Red Hat
- Expose kernel-level virtual devices to userspace using VFIO API

Resources

- http://vfio.blogspot.com
  - IOMMU Groups, inside and out
  - VFIO GPU How-To Series
- vfio-users mailing list
- #vfio-users on freenode

http://awilliam.github.io/presentations/KVM-Forum-2016

Questions?

Alex Williamson / alex.williamson@redhat.com