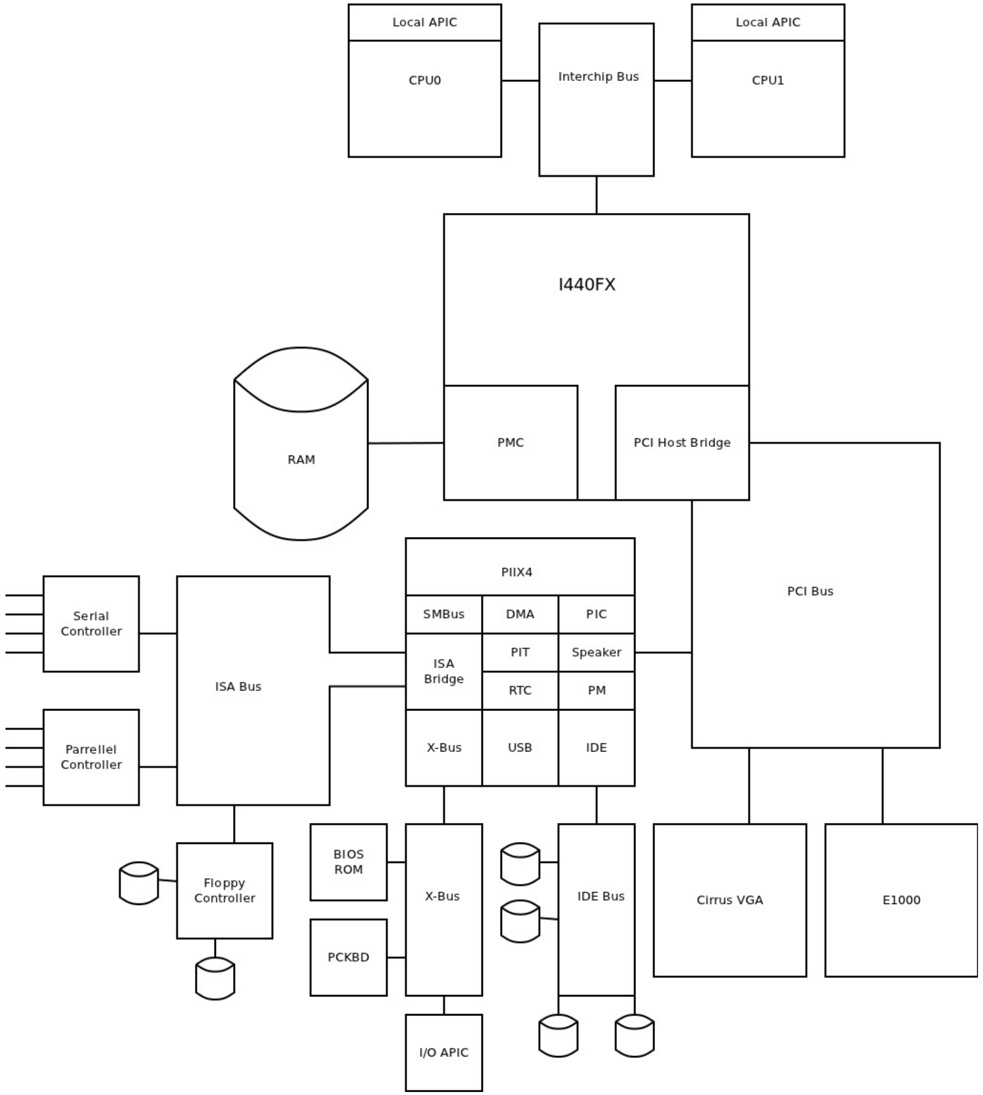
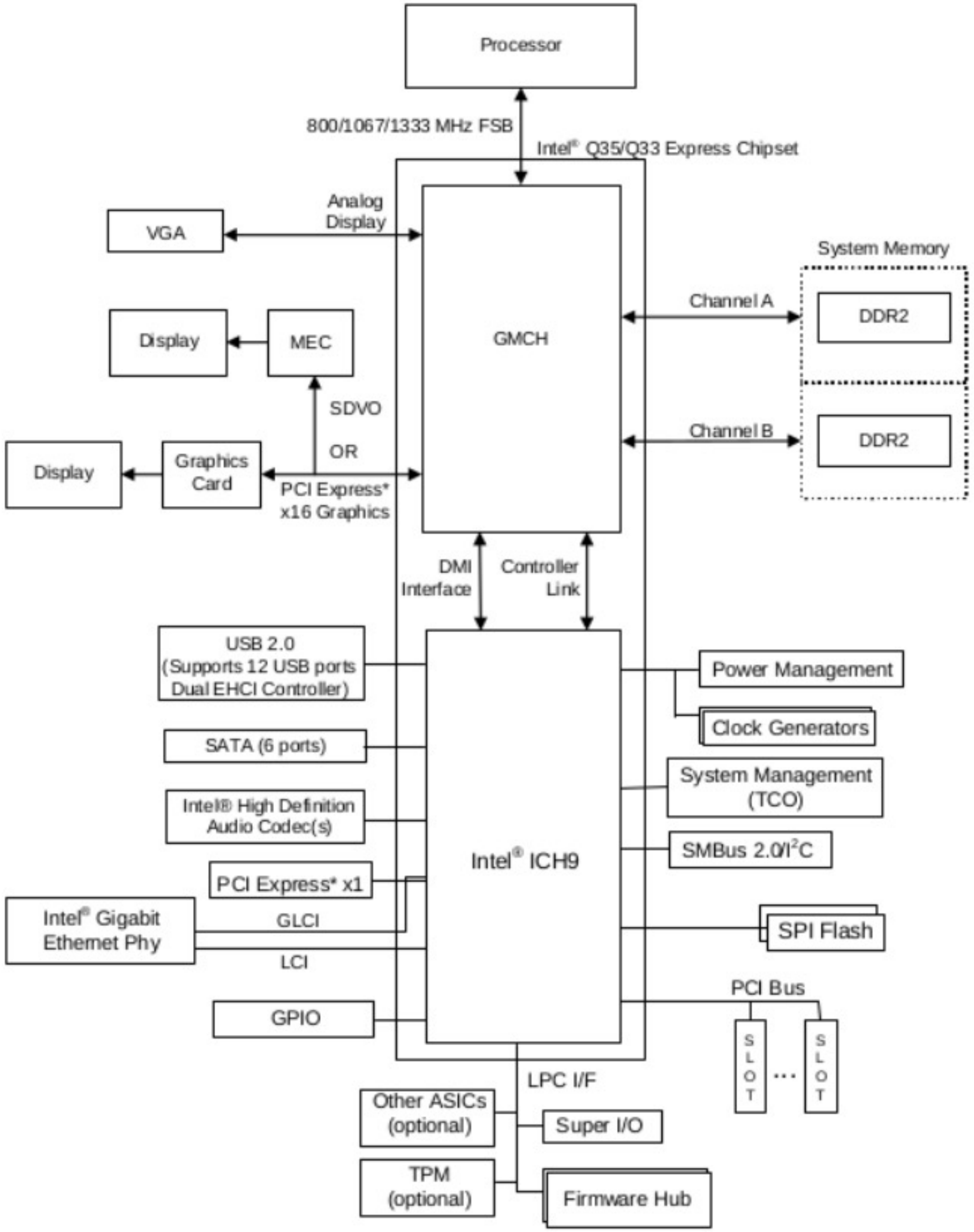
KVM & QEMU
IO model

Why the virtualization?

PC Architecture



I440FX+PIIX4 (1996)



Q35 (2007)

Topology of I440FX vs. Q35

- Q35 has IOMMU
- Q35 has PCIe
- Q35 has Super I/O chip with LPC interconnect
- Q35 has 12 USB ports
- Q35 SATA vs. PATA(Paraller Advanced Technology Attachment, aka. IDE)

How to virtualize?

- CPU Virtualization
  - Trap and Emulate Model
  - Virtualization technique, VMX Root/Non-Root Operation, VMM and Guest OS, VMCS … etc.

- Memory Virtualization: Extended Page Tables (EPT)
  - EPT implement one more page table hierarchy
  - MMU virtualize, EPT translation, Memory Operation,… etc.

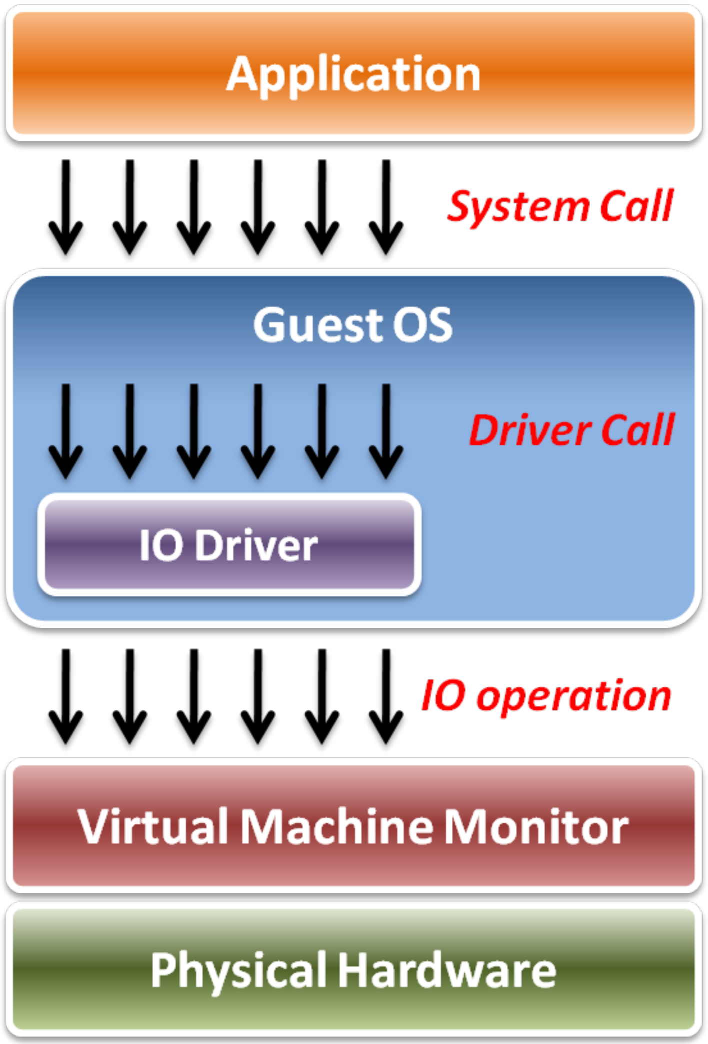
- IO Virtualization:
  - Implement DMA remapping in hardware
  - Hardware Page Walk, Translation Caching

IO virtualization

- Goal :
  - Share or create I/O devices for virtual machines.
- Two types of IO subsystem architecture :
  - Port Mapped IO (PMIO)
    - Port-mapped IO uses a special class of CPU instructions specifically for performing IO.
  - Memory Mapped IO (MMIO)
    - Memory Mapped IO uses the same address bus to address both memory and IO devices, and the CPU instructions used to access the memory are also used for accessing devices.
- Traditional IO techniques :
  - Direct memory Access (DMA)
  - PCI / PCI Express

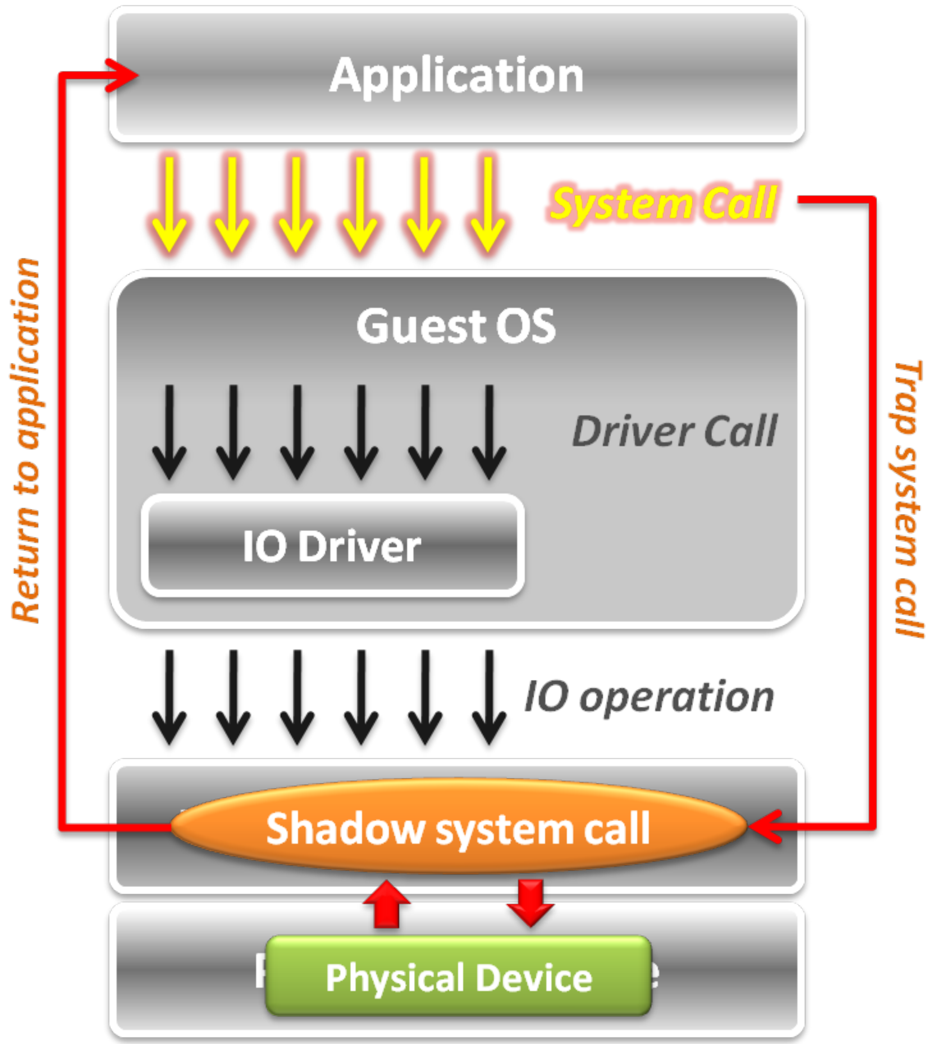
IO virtualization

- Implementation Layers :
  - System call
    - The interface between applications and guest OS.
  - Driver call
    - The interface between guest OS and IO device drivers.
  - IO operation
    - The interface between IO device driver of guest OS and virtualized hardware (in VMM).



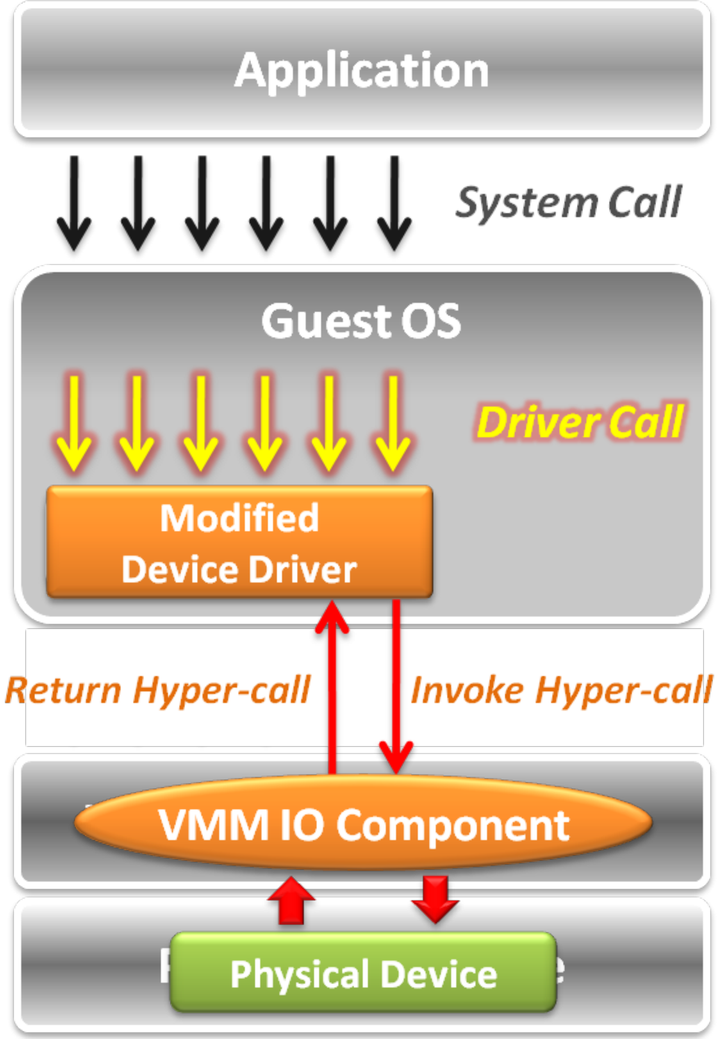
IO virtualization

- In system call level :
  - When an application invokes a system call, the system call will be trapped to VMM first.
  - VMM intercepts system calls, and maintains shadowed IO system call routines to simulate functionalities.
  - After simulation, the control goes back to the application in gust OS.



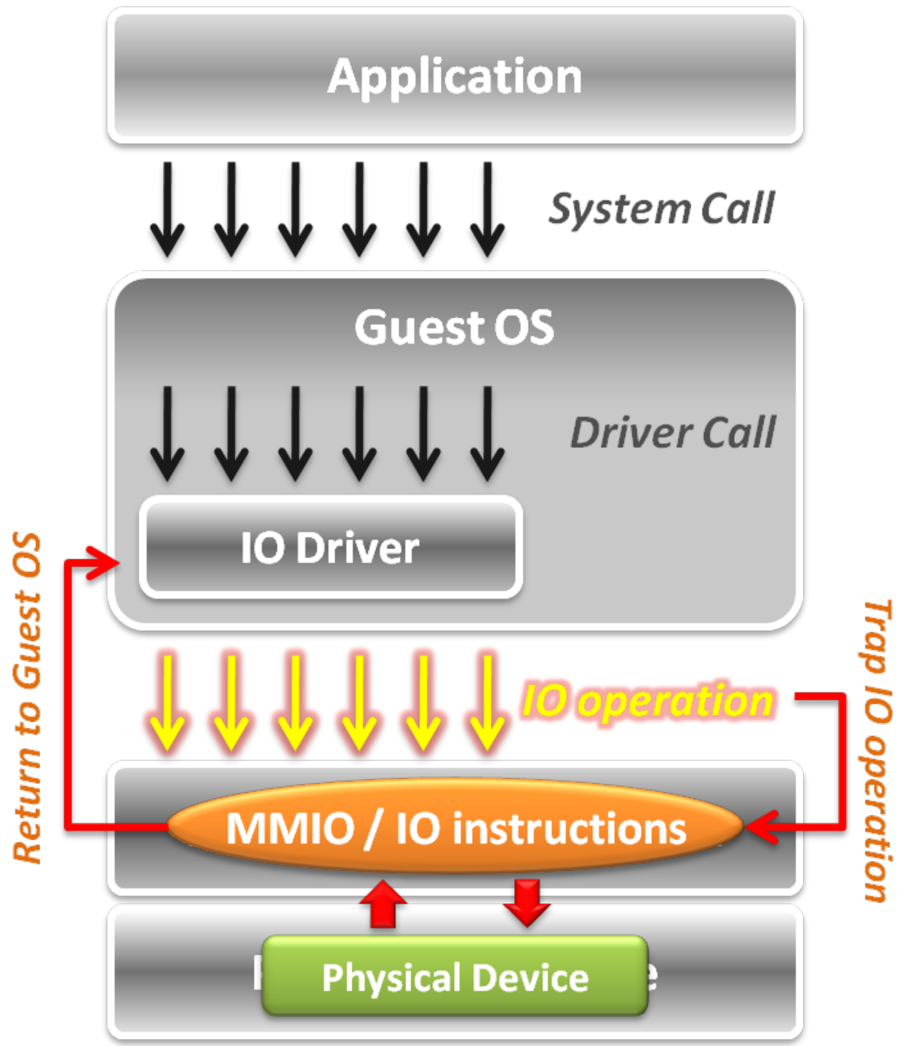
IO virtualization

- In device driver call level :
  - Adopt the **para-virtualization** technique, which means the IO device driver in guest OS should be modified.
  - The IO operation is invoked by means of hyper-call between the modified device driver and VMM IO component.

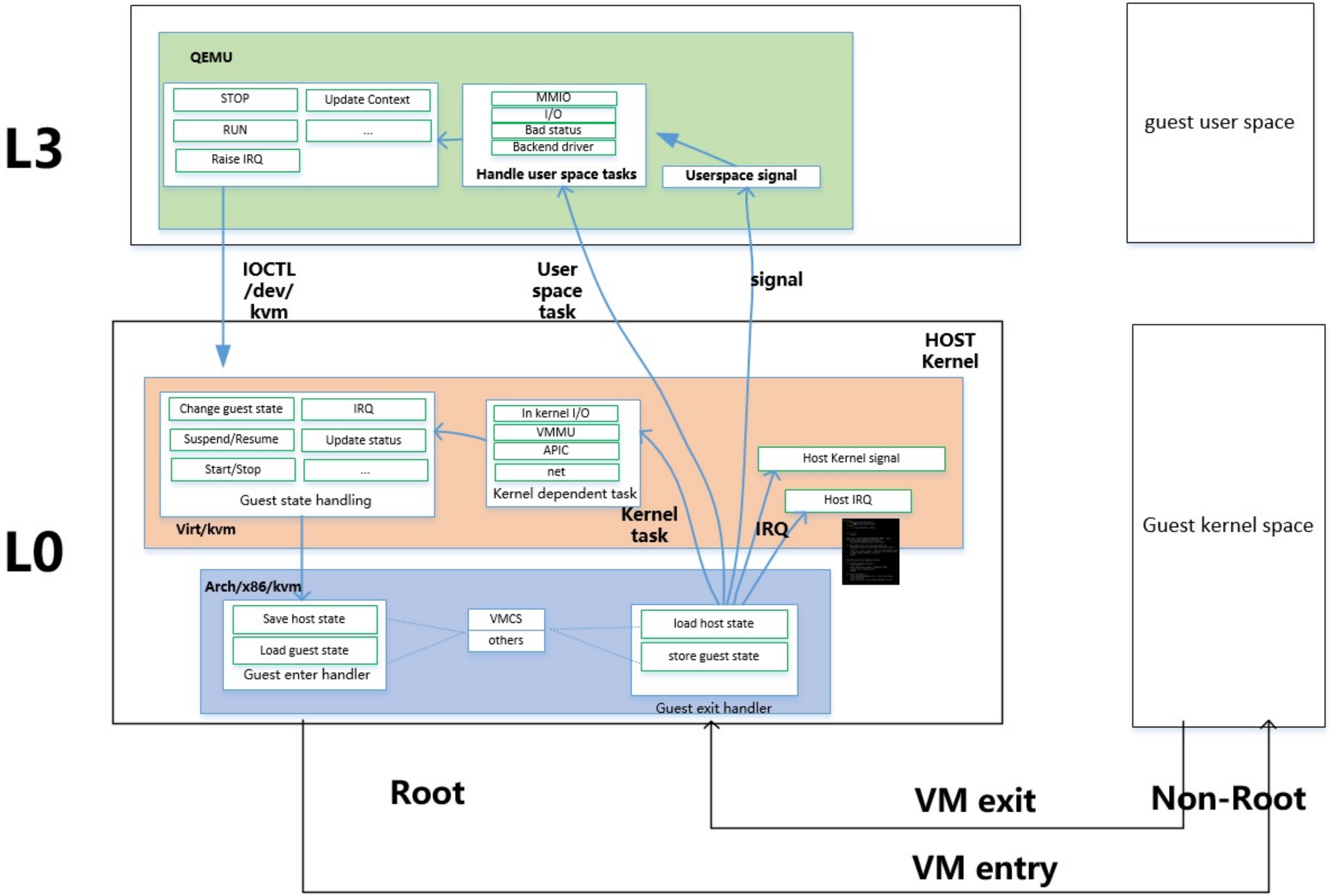


IO virtualization

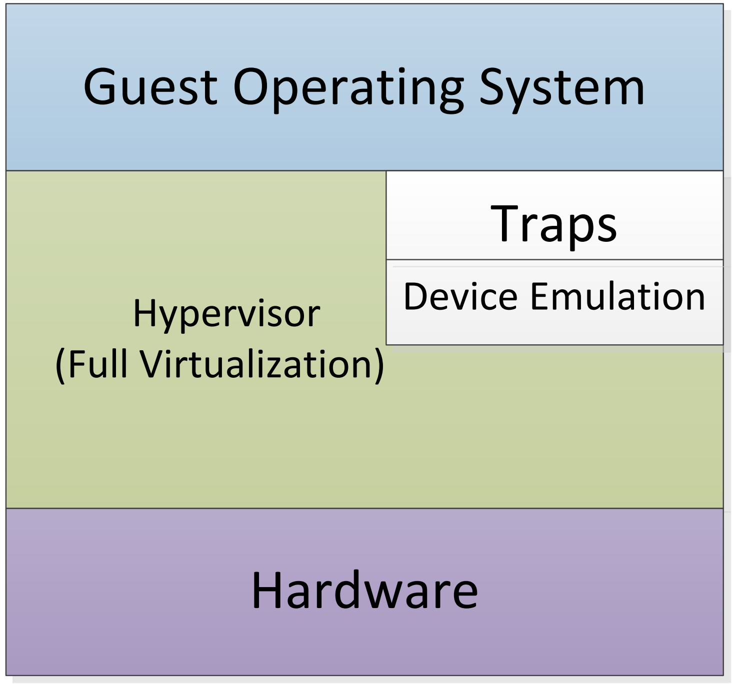
- In IO operation level(**full virtualization**),
  - Port mapped IO
    - Special input/output instructions with special addresses.
    - The IO instructions are privileged .
    - On X86, I/O bitmaps in VM-execution control field can be used to configure which port execution(VM) would cause VM-Exit.
  - Memory mapped IO
    - Loads/stores to specific region of real memory are interpreted as command to devices.
    - The memory mapped IO region is protected.
    - These range of memory will NOT be mapped by QEMU/KVM, thus when VM touch this memory, VM-Exit happens, then KVM will route this Exit to QEMU for further operation.



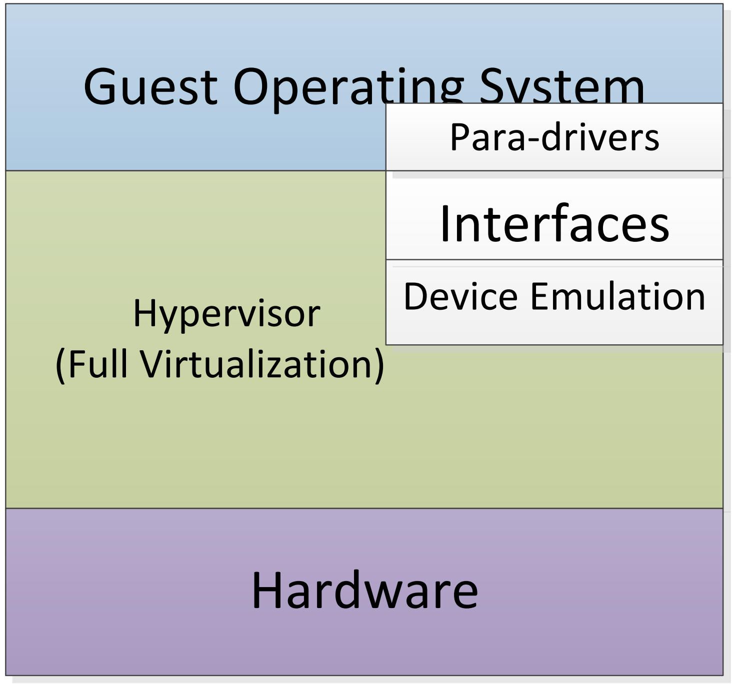
QEMU/KVM Overview



IO virtualization-Overview



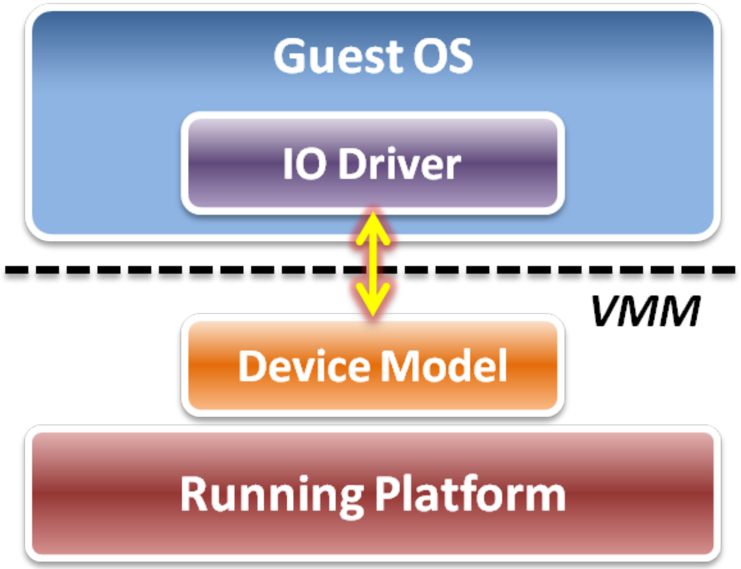
Full virtualization



Para-virtualization

Device model-Full virtualization

- Focus on IO operation level implementation.
  - This is an approach of full virtualization.
- Logic relation between guest OS and VMM :
  - VMM intercepts IO operations from guest OS.
  - Pass these operations to device model on a running platform.
  - Device model needs to emulate the IO operation interfaces.
  - Port mapped IO
  - Memory mapped IO
  - DMA
  - ... etc.

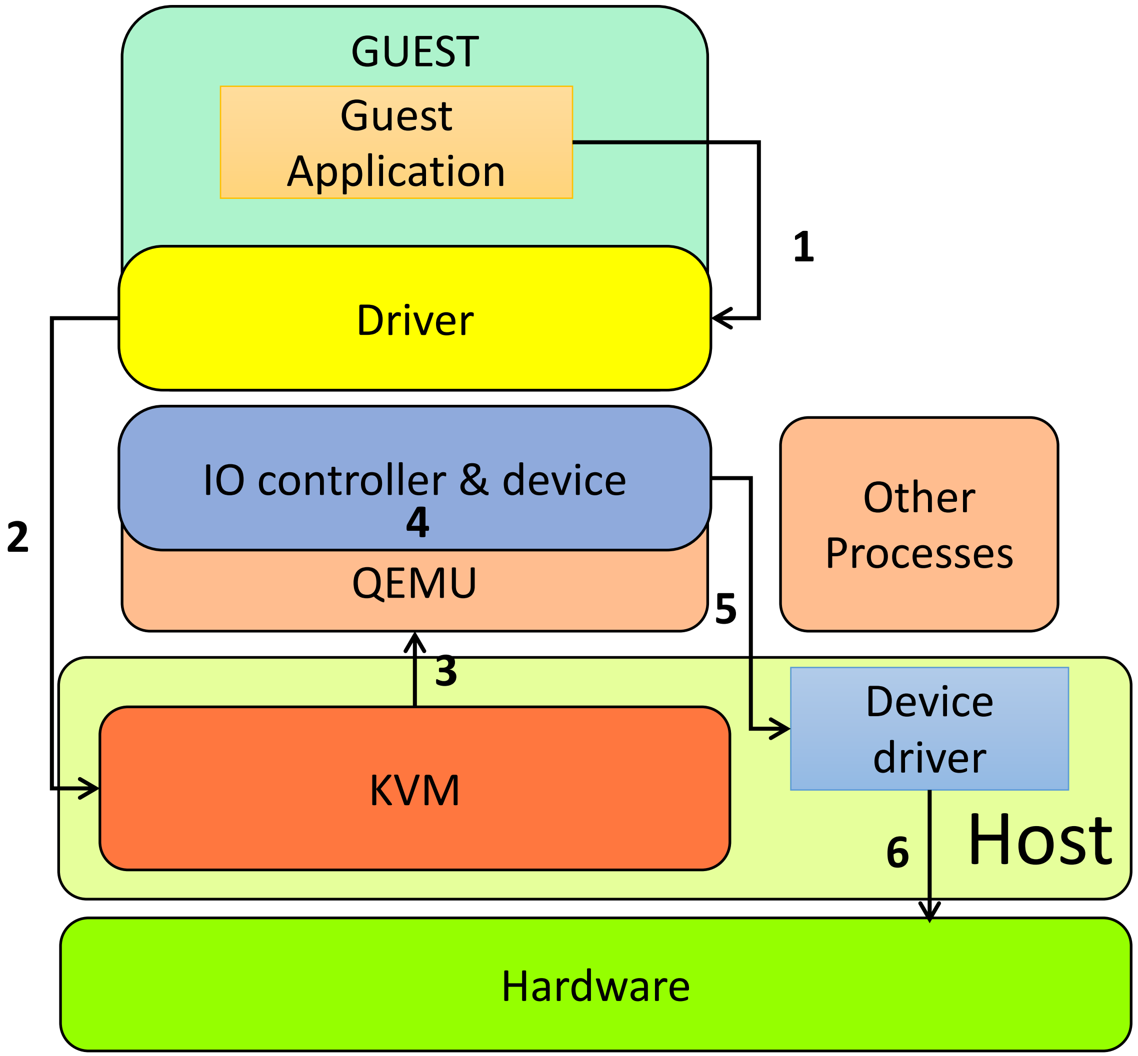


Device virtualization

IO device types :

- Dedicated device
  - Ex : displayer, mouse, keyboard …etc.
- Partitioned device
  - Ex : disk, tape …etc
- Shared device
  - Ex : network card, graphic card …etc.
- Nonexistent physical device
  - Ex : virtual device …etc.

Full virtualization



1. Guest application issued a system call to read data from Disk
2. Guest OS driver will start I/O operation, then trap to HOST.
3. KVM analysis the Exit reason, then return to QEMU.
4. QEMU process analysis the Exit reason, call corresponding block device back-end driver to handle this I/O.
5. QEMU call host device driver to accomplish this I/O operation.
6. Host device driver access the real hardware to get the data.

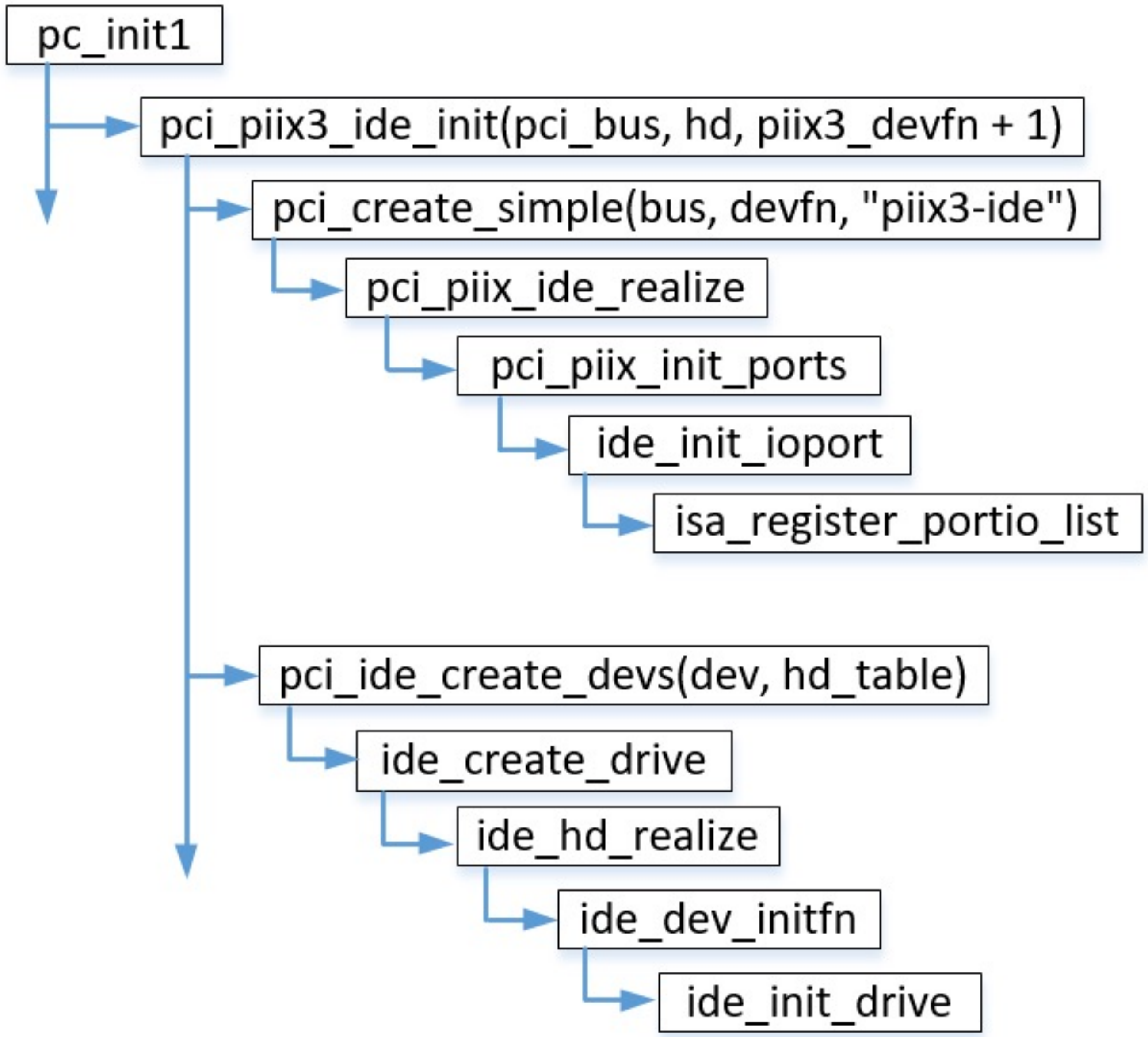
An example: Block device

How to emulate a IDE block device

- *qemu-system-x86_64 -enable-kvm -m 4096 -hda vdisk.img -vga std -smp 32 -vnc :1*

Let's see what does QEMU actually did to create an IDE block device

An example: Block device



In main, qemu would call *configure_blockdev* to init a block-device backend.

Then in vm machine board init phase, pci_piix3_ide_init will create a IDE controller which name is “piix3-ide ”, and hook this divice to pci_bus, then pci_piix_ide_realize is invoked, to init piix3-ide’s configuration space, pci bar, register a reset function, and register ioports(0x1f0, 0x3f6).

And pci_ide_create_devs will create a emulated hard-disk device and connect it to ide controller.

An example: Block device

VM hardware info

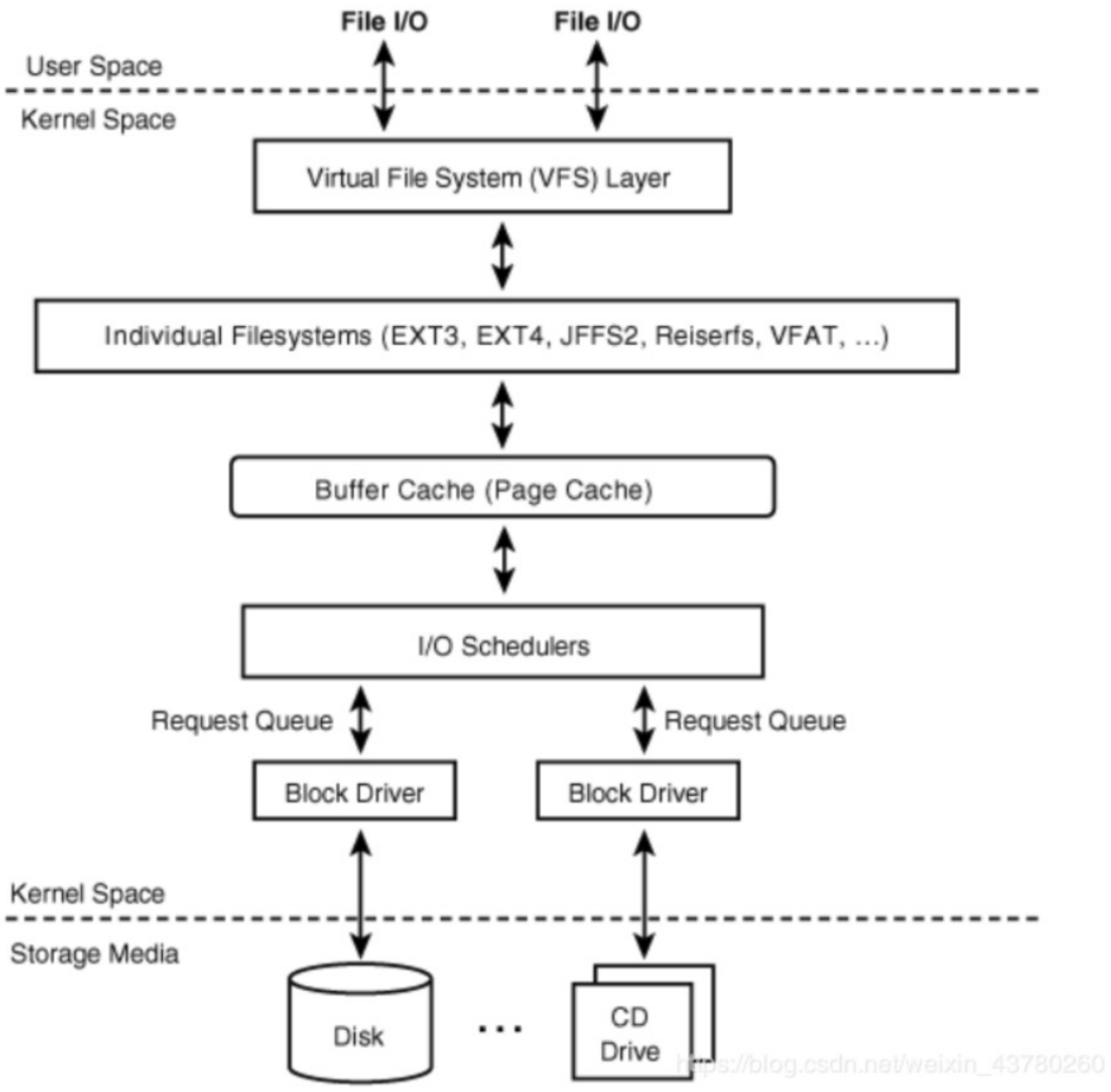
```
(qemu) info pci
  Bus  0, device   0, function 0:
    Host bridge: PCI device 8086:1237
      PCI subsystem 1af4:1100
      id ""
  Bus  0, device   1, function 0:
    ISA bridge: PCI device 8086:7000
      PCI subsystem 1af4:1100
      id ""
  Bus  0, device   1, function 1:
    IDE controller: PCI device 8086:7010
      PCI subsystem 1af4:1100
      BAR4: I/O at 0xc040 [0xc04f].
      id ""
  Bus  0, device   1, function 3:
    Bridge: PCI device 8086:7113
      PCI subsystem 1af4:1100
      IRQ 9.
      id ""
  Bus  0, device   2, function 0:
    VGA controller: PCI device 1234:1111
      PCI subsystem 1af4:1100
      BAR0: 32 bit prefetchable memory at 0xfd000000 [0xfdffffff].
      BAR2: 32 bit memory at 0xfebf0000 [0xfebf0fff].
      BAR6: 32 bit memory at 0xffffffffffffffff [0x0000fffe].
      id ""
  Bus  0, device   3, function 0:
    Ethernet controller: PCI device 8086:100e
      PCI subsystem 1af4:1100
      IRQ 11.
      BAR0: 32 bit memory at 0xfebc0000 [0xfebdffff].
      BAR1: I/O at 0xc000 [0xc03f].
      BAR6: 32 bit memory at 0xffffffffffffffff [0x0003fffe].
      id ""
```

```
root@test-Standard-PC-i440FX-PIIX-1996:/tmp/guest-hacakg# lshw -class disk
  *-disk
       description: ATA Disk
       product: QEMU HARDDISK
       physical id: 0.0.0
       bus info: scsi@0:0.0.0
       logical name: /dev/sda
       version: 2.5+
       serial: QM00001
       size: 16GiB (17GB)
       capabilities: partitioned partitioned:dos
       configuration: ansiversion=5 logicalsectorsize=512 sectorsize=512 signature=3ef1a6df
```

An example: Block device

A brief introduction to Block device

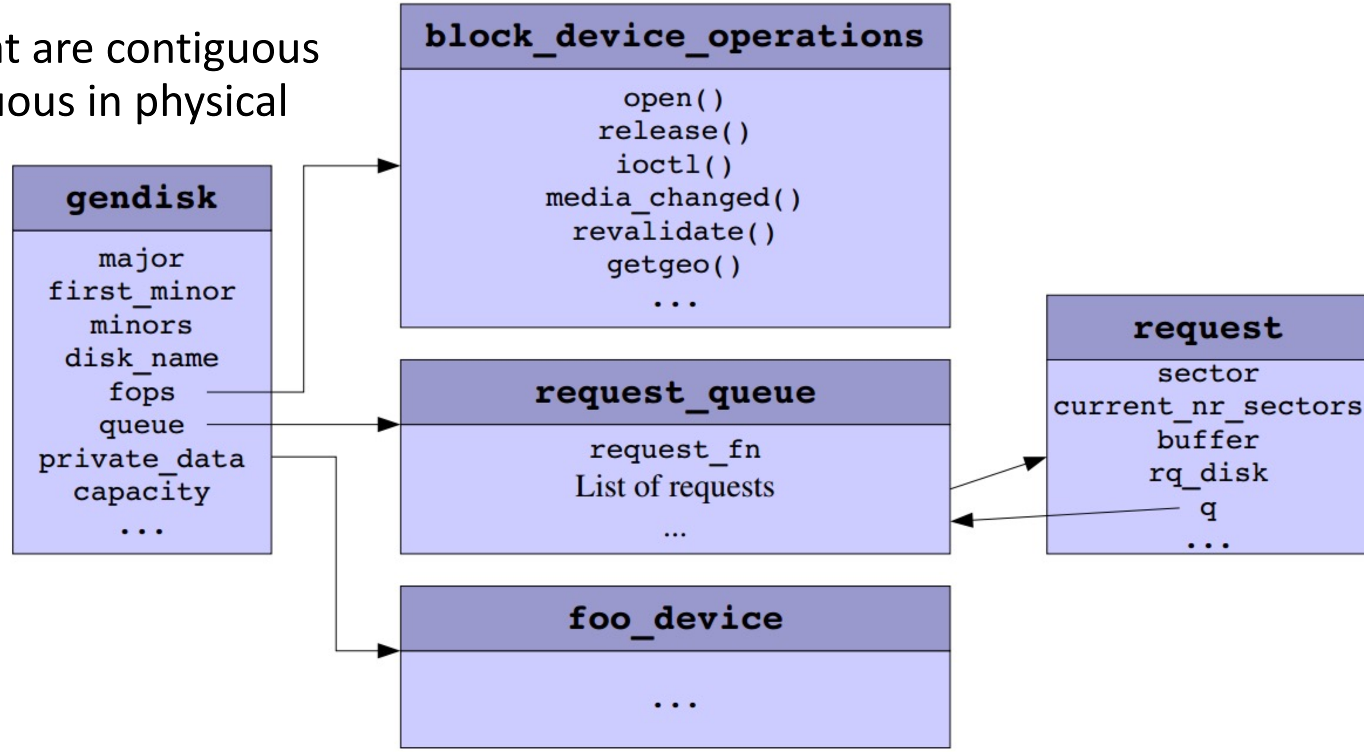
- An user application can use a block device
  - Through a filesystem, by reading, writing or mapping files
  - Directly, by reading, writing or mapping a device file representing a block device in /dev
- In both cases, the VFS subsystem in the kernel is the entry point for all accesses
  - A filesystem driver is involved if a normal file is being accessed
- The buffer/page cache of the kernel stores recently read and written portions of block devices
  - It is a critical component for the performance of the system
- I/O scheduling allows to
  - Merge requests so that they are of greater size
  - Reorder requests so that the disk head movement is as optimized as possible



An example: Block device

A brief introduction to Block device

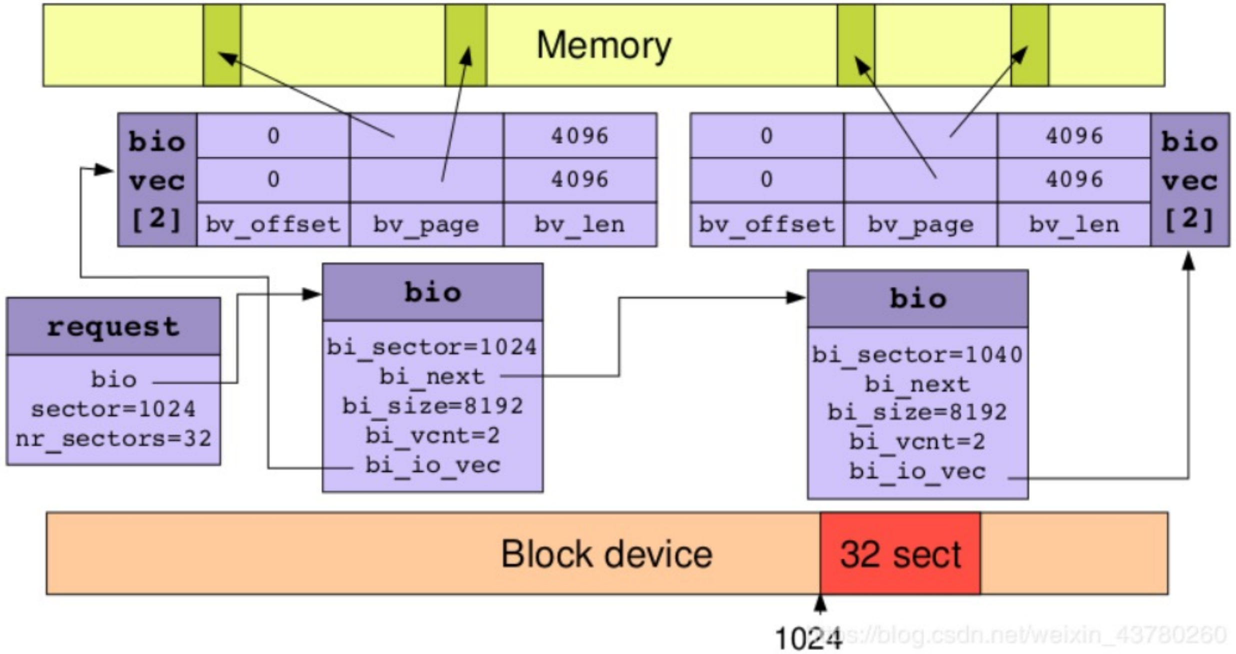
- struct gendisk is an abstraction of real disk
- Request_queue is a queue collecting requests, driver need to realize request_fn to consume these request.
- A request is composed of several segments, that are contiguous on the block device, but not necessarily contiguous in physical memory.
- A struct request is in fact a list of struct bio

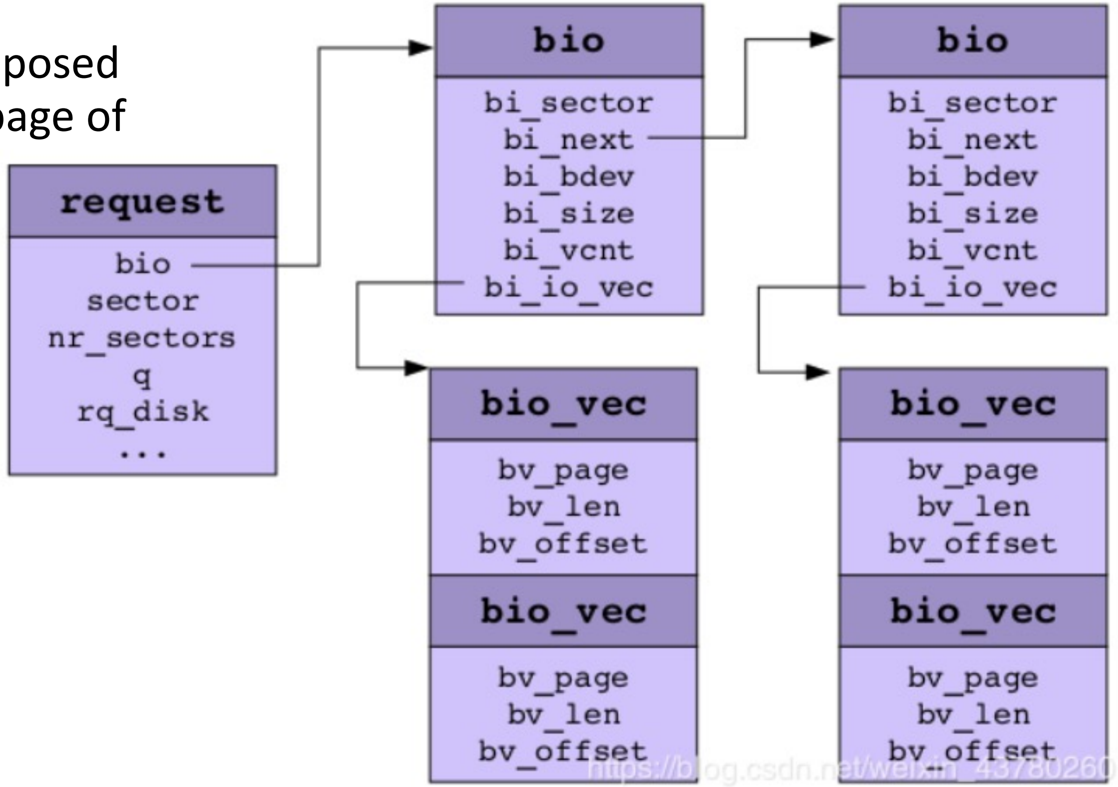


An example: Block device

A brief introduction to Block device

- A bio is the descriptor of an I/O request submitted to the block layer. BIOs are merged together in a ***struct request*** by the I/O scheduler.
- As a bio might represent several pages of data, it is composed of several ***struct bio_vec***, each of them representing a page of memory.



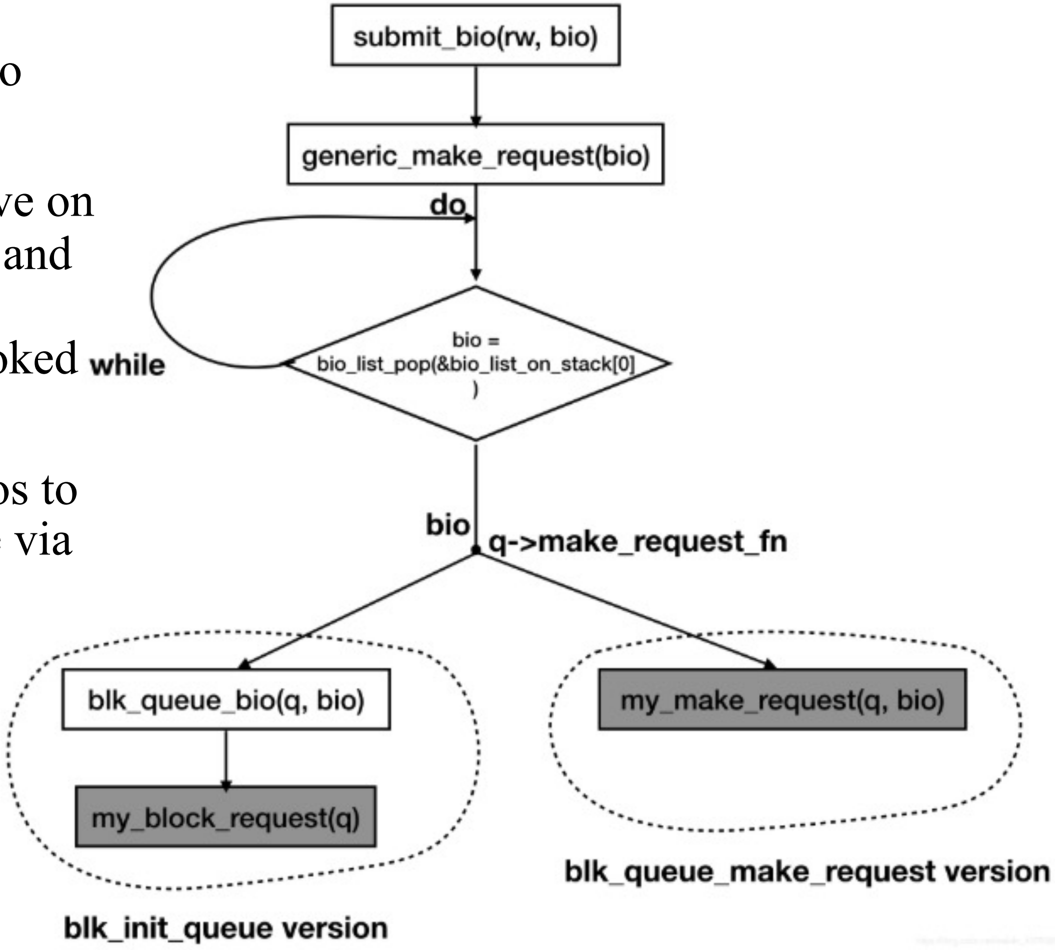


An example: Block device

A brief introduction to Block device

1. upper layer create a bio structure and use *submit_bio(rw, bio)* to process these bio
2. *generic_make_request* would check if *make_request_fn* is active on this task right now, if yes, append the bio at the end of bio_list and return, if no, it will pop bio from current->bio_list, and call q->make_request_fn, if using request verison, the function is hooked with *blk_queue_bio*(q, bio), or it's user defined
3. *blk_queue_bio* would rearrange the bio in the queue, merge bios to optimize the operation, and copy data from new bio into queue via *init_request_from_bio*(req, bio)
4. Then calling *__blk_run_queue* to process the queue.
5. Would eventually call *request_fn*, the function will do I/O

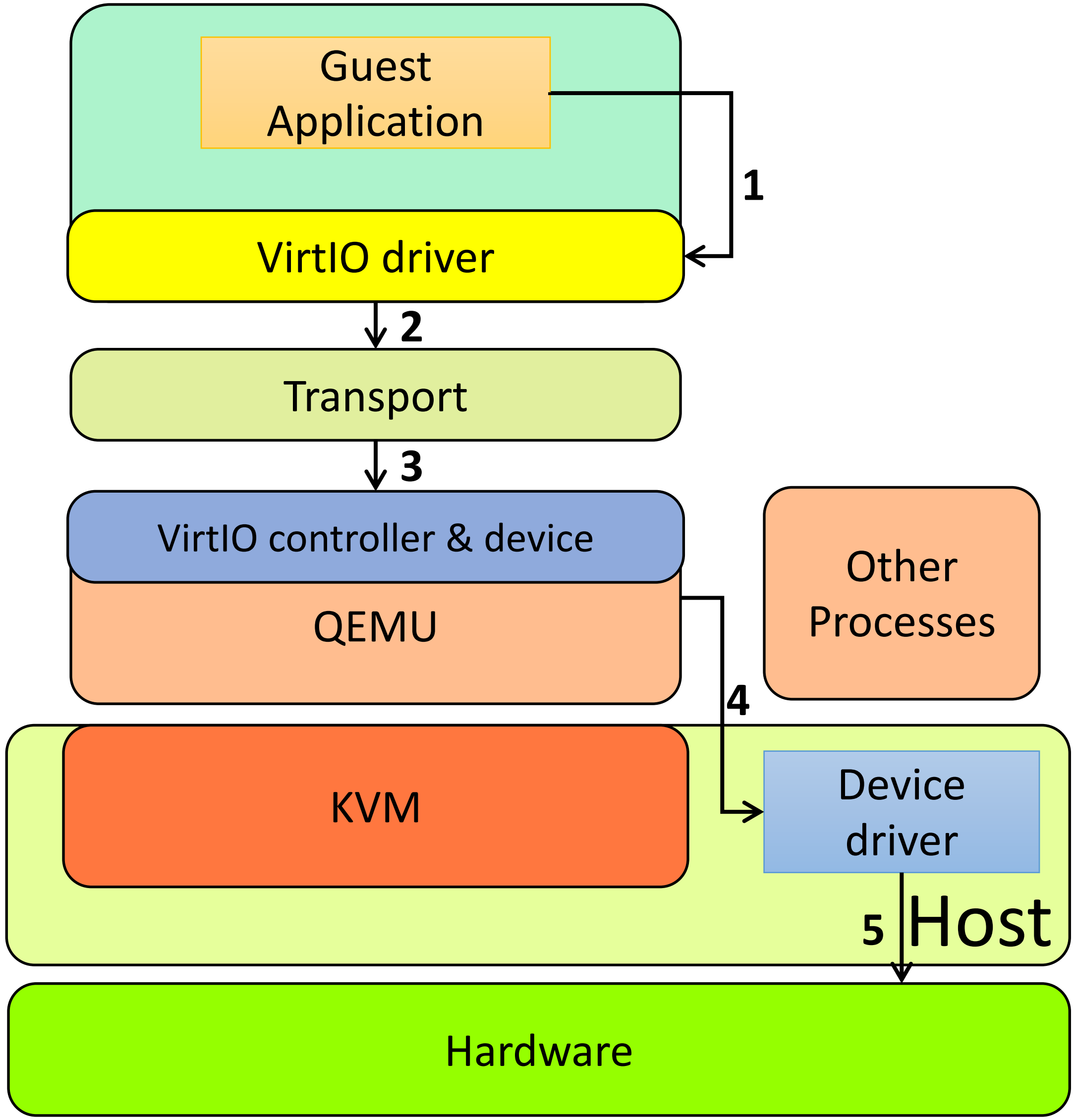
   operation to consume the requests.



An example: Block device

1. When guest app want to write data into it's "disk", after all the way from File I/O -> VFS -> EXT4 -> BufferCache -> I/O scheduler -> Block driver, guest OS driver will do IO operation to consume the request.
2. IO command operation will cause VMExit into Host because guest ide controller's IO port is not mapped in the MMU stage 2 page table.
3. Then KVM will analysis the VM Exit reason, find he cannot handle this, will copy port and data info into vcpu structure, then set exit_reason to KVM_EXIT_IO, now, return back to QEMU.
4. QEMU will also analysis the VM Exit reason, then calling *kvm_handle_io* to process this "Trap", use *address_space_rw* to manipulate IO address space(another is Memory address space), then QEMU translate the address space to flat view, find the coordinate memory region, use the ops register in the memory region to complete the rw operation, in this case, is *ide_ioport_write* which belong to IDE driver.
5. IDE driver have many other functions stored in ide_cmd_table, then dispatch io operation to *cmd_write_pio,* this routine will finally goes to Host OS system call *pwrite.*
6. Host system will do it again like what have been done in VM in step 1. and host system definitely knows where to write because QEMU block device back-end driver provide a file to VM as a disk, so host system knows where this file is mapped on the RAM.

KVM with Virt-IO



1. Guest application issued a system call to read data from disk
2. VirtIO driver will transport the request into a memory.
3. VirtIO back-end driver will fetch the request from that memory.
4. VirtIO back-end driver will issued a system call to read data from Host disk.
5. Host device driver would finish the journey.

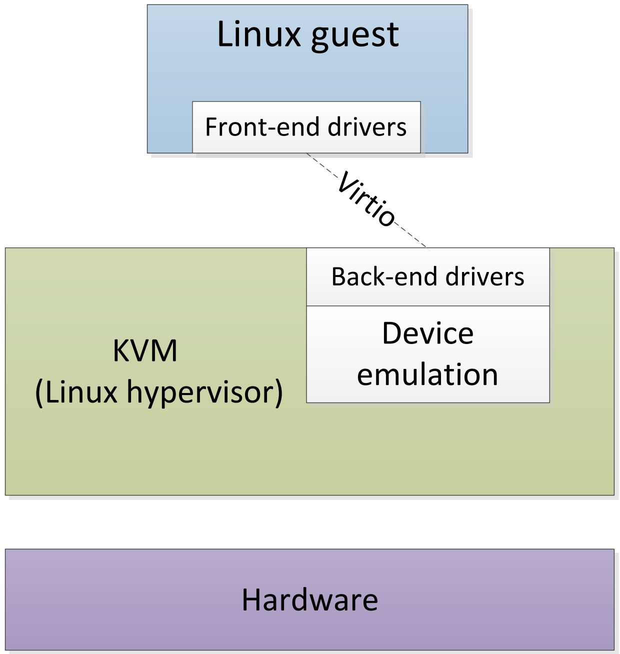
Why Virt-IO

- Too many VM Exit in full virtualization scheme, causing performance drop in block device and network device.
- QEMU is emulating at a lowest level of the conversation(like block device) in full virtualization scheme, which is inefficient and highly complicated.
- Memory copy between VM and Host is a waste.
- A new implementation is needed.

What is Virt-IO

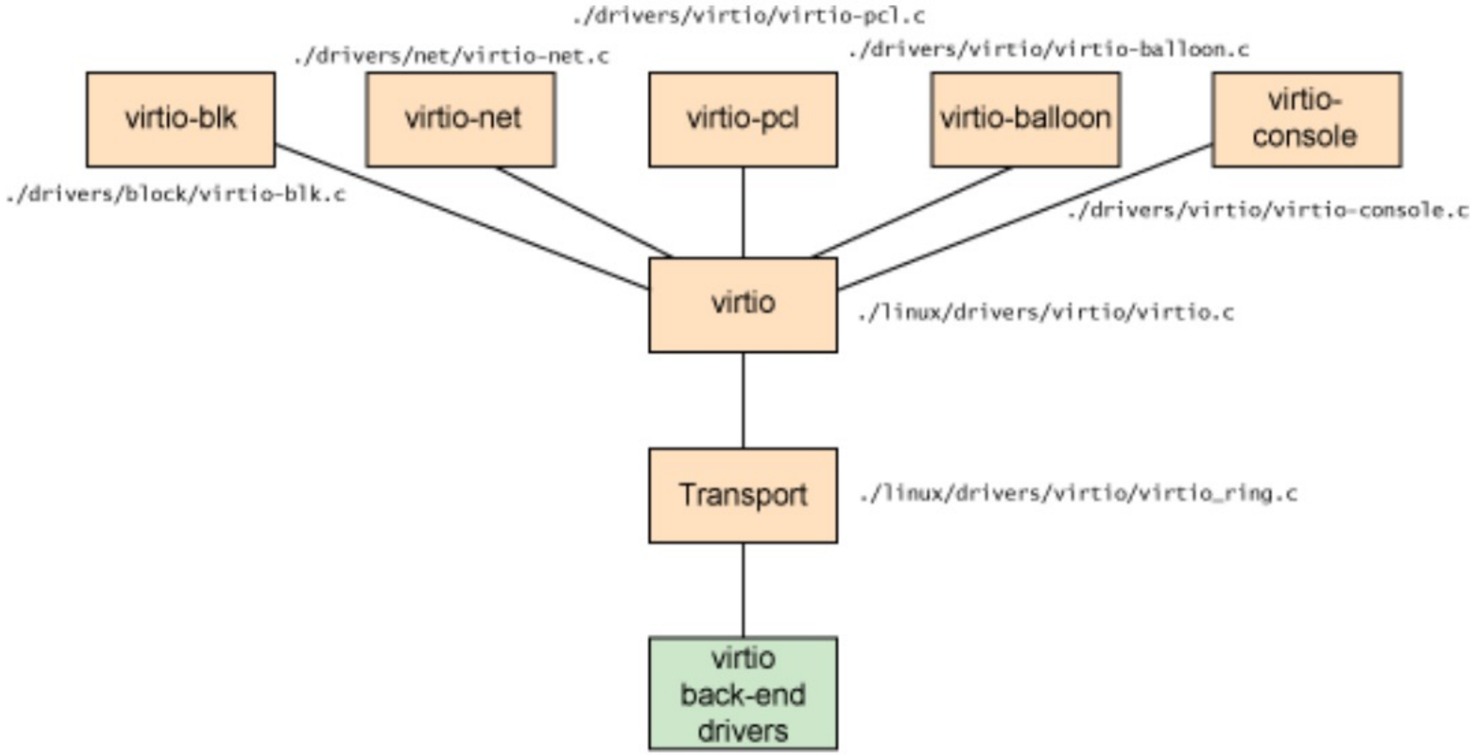
Virtio is a Linux IO virtualization standard for network and disk device drivers and cooperates with the hypervisor:

- It provides a set of APIs and structures for making virtio devices.
- The host implementation is in userspace - qemu, so no driver is needed in the host
- Only the guest's device drivers aware the virtual environment.
- This enables guests to get high performance network and disk operations, and gives most of the performance benefits of para-virtualization.



What is Virt-IO

High-level architecture of the virtio framework, as showned:



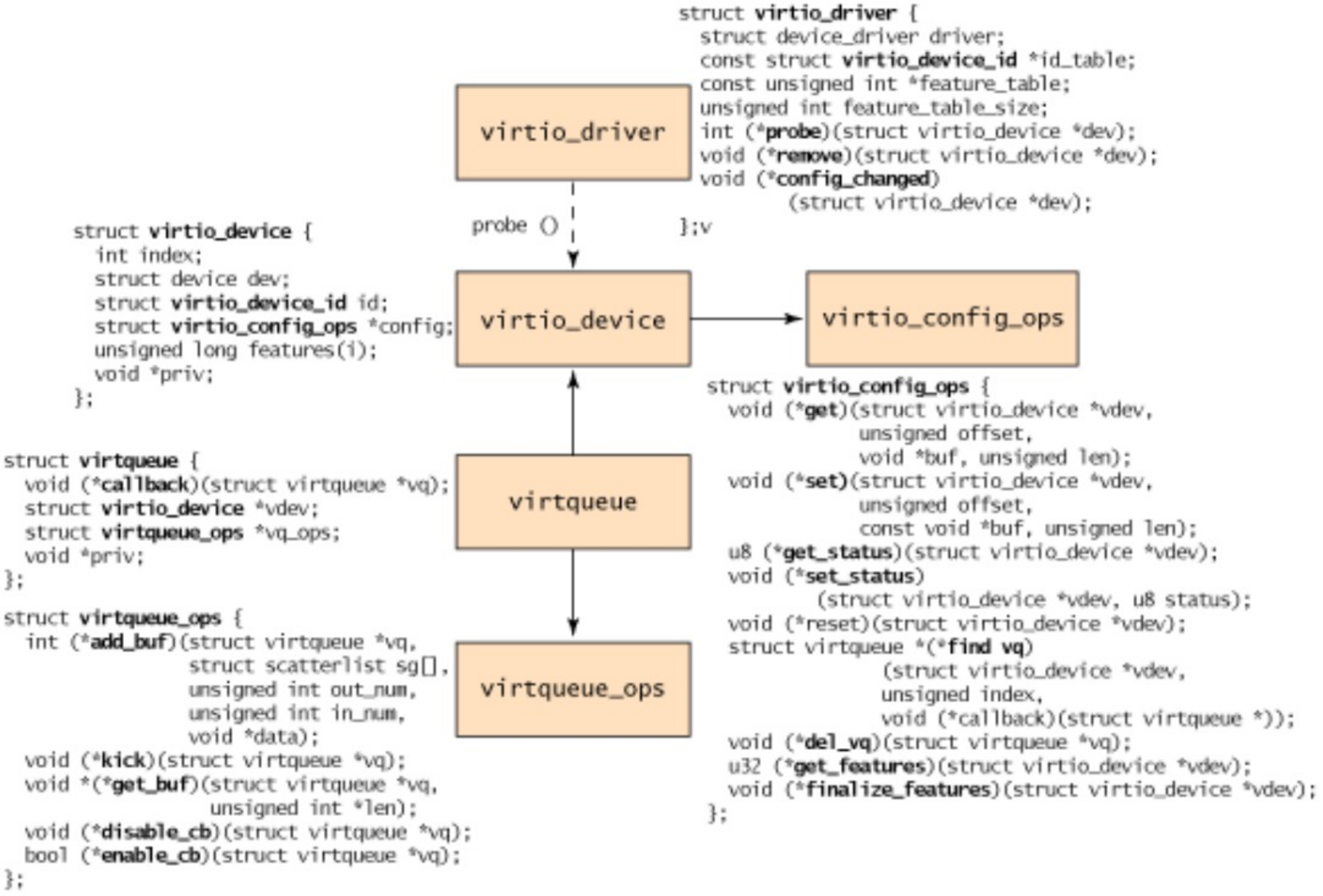
- Have 5 front-end drivers: virtio-blk, virtio-net, virtio-pci, virtio-balloon, virtio-console, and each front-end driver has a back-end driver in hypervisor.

What is Virt-IO

- Virtio is a virtual queue interface that conceptually attaches front-end drivers and back-end drivers. Implemented as rings called Virtio_ring, and it's a memory mapped region between QEMU and Guest.
- Transport is an abstraction layer about how front-end and back-end communicate, probing and configuration for all virtual devices. virtio can use various different buses, such as PCI, MMIO, Channle I/O.

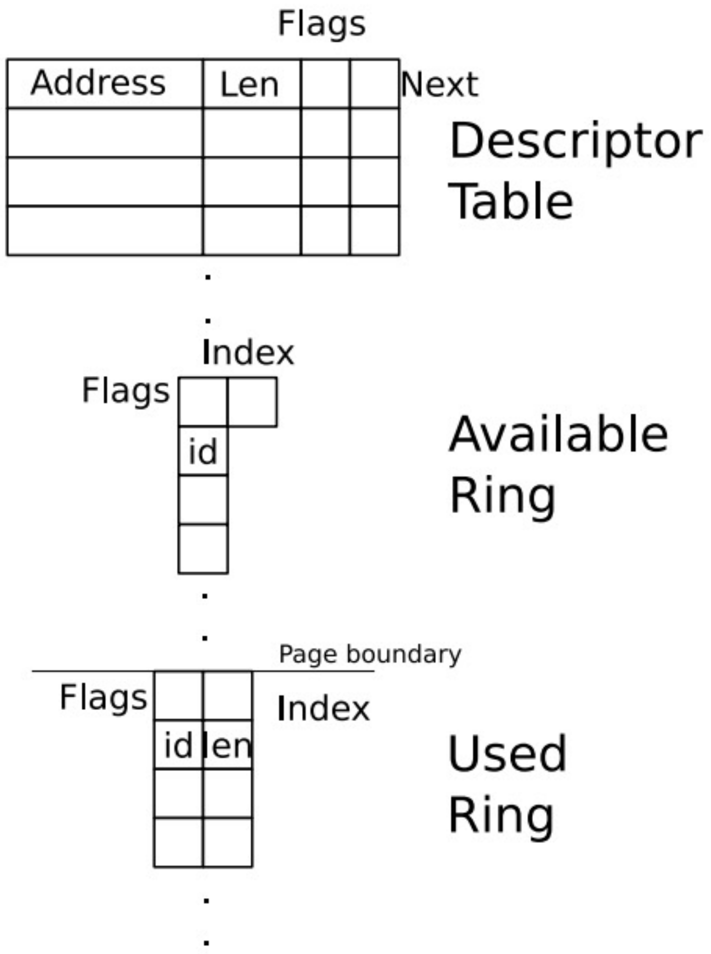
What is Virt-IO(cont.)

Object hierarchy of the virtio front end



What is Virt-IO

The virtio_ring consists of three parts:




```c
/* Virtio ring descriptors: 16 bytes.  These can chain together via "next". */
struct vring_desc {
	/* Address (guest-physical). */
	__virtio64 addr;
	/* Length. */
	__virtio32 len;
	/* The flags as indicated above. */
	__virtio16 flags;
	/* We chain unused descriptors via this, too */
	__virtio16 next;
};

struct vring_avail {
	__virtio16 flags;
	__virtio16 idx;
	__virtio16 ring[];
};

/* u32 is used here for ids for padding reasons. */
struct vring_used_elem {
	/* Index of start of used descriptor chain. */
	__virtio32 id;
	/* Total length of the descriptor chain which was used (written to) */
	__virtio32 len;
};

struct vring_used {
	__virtio16 flags;
	__virtio16 idx;
	struct vring_used_elem ring[];
};
```

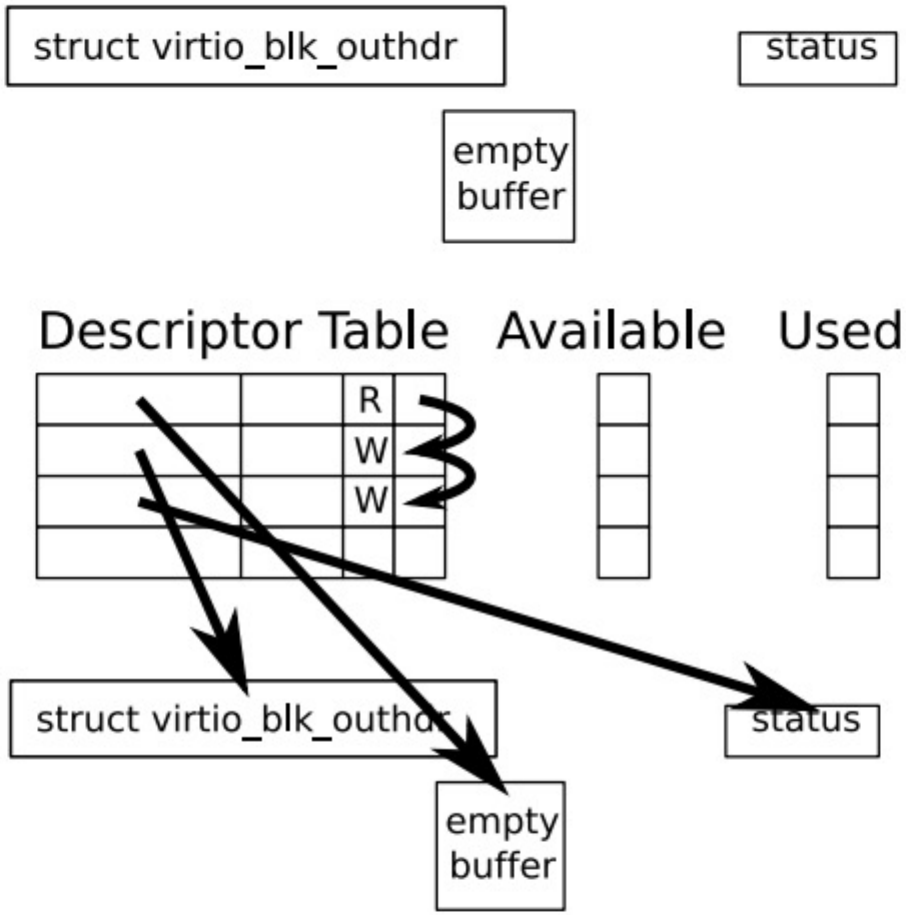
What is Virt-IO

- **Descriptor table**: Each descriptor contains the guest-physical address of the buffer, its length, an optional ’next’ buffer for chaining, and two flags: one to indicate whether the next field is valid and one controlling whether the buffer is read-only or write only. In short, it’s used to describing the buffer.
- **Available ring** consists of a free-running index, an interrupt suppression flag, and an array of indices into the descriptor table (representing the heads of buffers). Supplied by the driver to the device.
- **Used ring** is similar to the available ring, but is written by the host as descriptor chains are consumed.

What is Virt-IO

Let's see a brief example(blk read):

1. We have a empty buffer for the date to read into, a struct virtio_blk_outhdr for request metadata and a status byte to represent a single request.

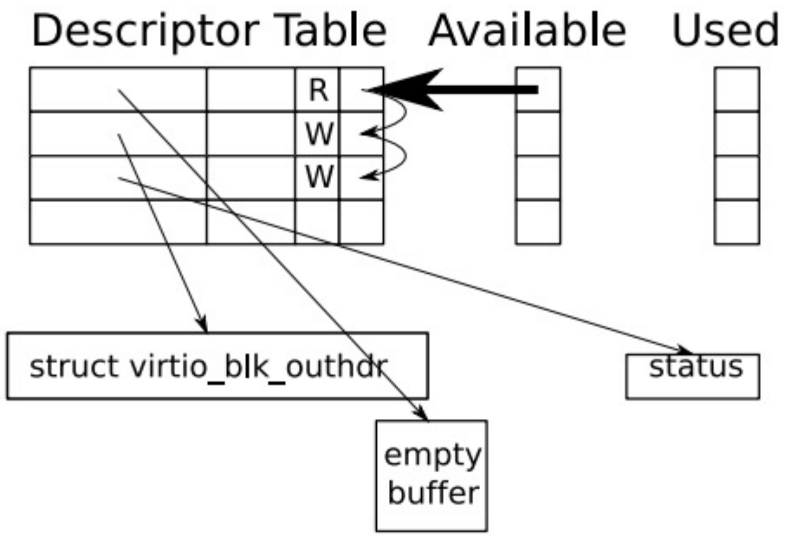


2. We put(add_buf) these three parts of our request into three free entries of the descriptor table and chain them together.

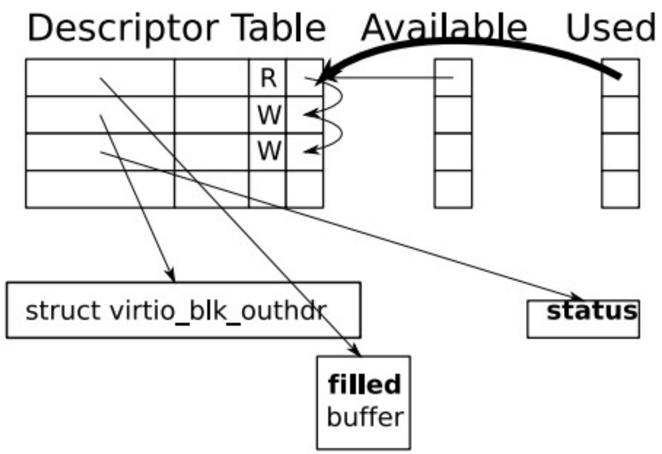
What is Virt-IO

Let’s see a brief example(blk read):

3. placing the index of the descriptor head into the “available” ring, issuing a memory barrier, then incrementing the available index. A “kick” is issued to notify the host that a request is pending

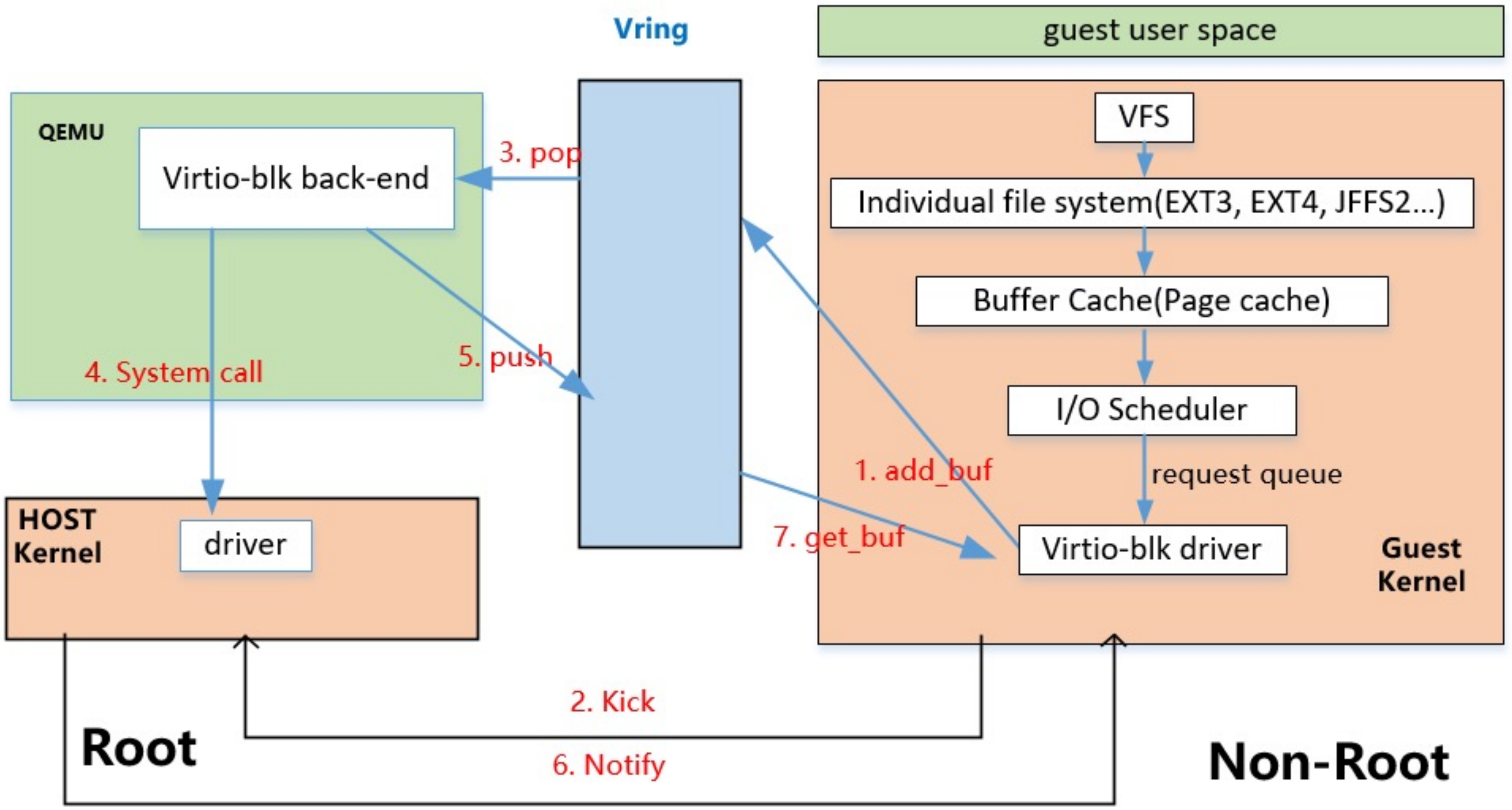


4. Host pop the requests from the available ring, after process the requests, host fill the buffer and update the status byte. At this point the descriptor head is push in the “used” ring and the guest is notified.



5. Now guest can withdraw(get_buf) the request result from the vring.

KVM with Virt-IO



Another example: Virtual VGA

Virtual VGA is virtual video graphics array:

- A PIC device.
- Bar0 is a frame buffer to store pixel data.
- Bar2 is a 4K MMIO region, vga ioports(0x3c0 -> 0x3df) is 1:1 remapped at 0x400-0x41f

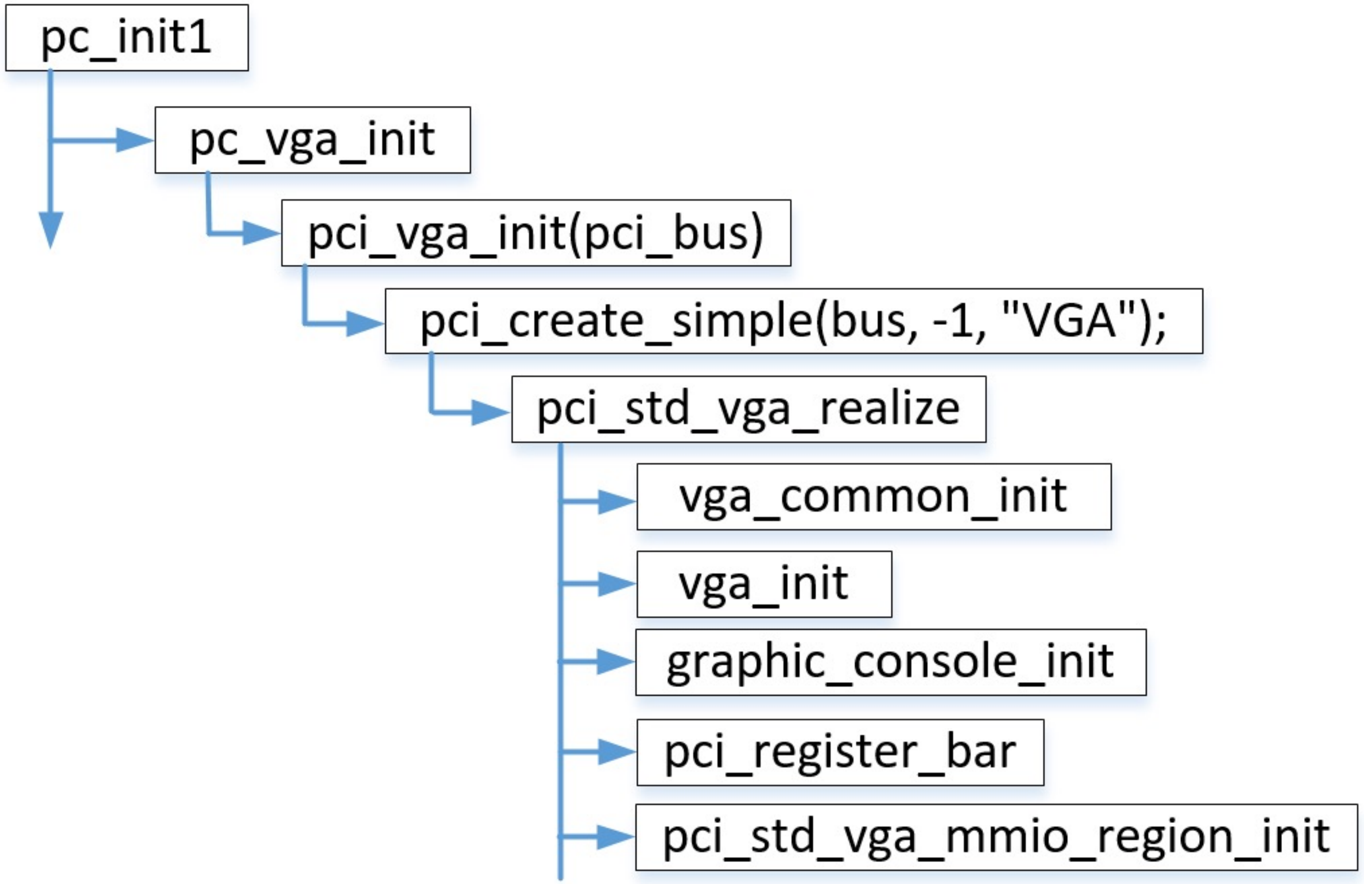
Another example: Virtual VGA

How to emulate a VGA device

- *qemu-system-x86_64 -enable-kvm -m 4096 -hda vdisk.img -vga std -smp 32 -vnc :1*

Let’s see what does QEMU actually did to create a vga std device

Another example: Virtual VGA



In *pc_init1*, *pc_vga_init* is called to create virtual vga device. If there is PCI bus, qemu would create a pci variant VGA.

pci_std_vga_realize is to realize a pci vga device. *vga_common_init* will “malloc” memory for vga vram and init VGACommonState, and start dirty log vram.

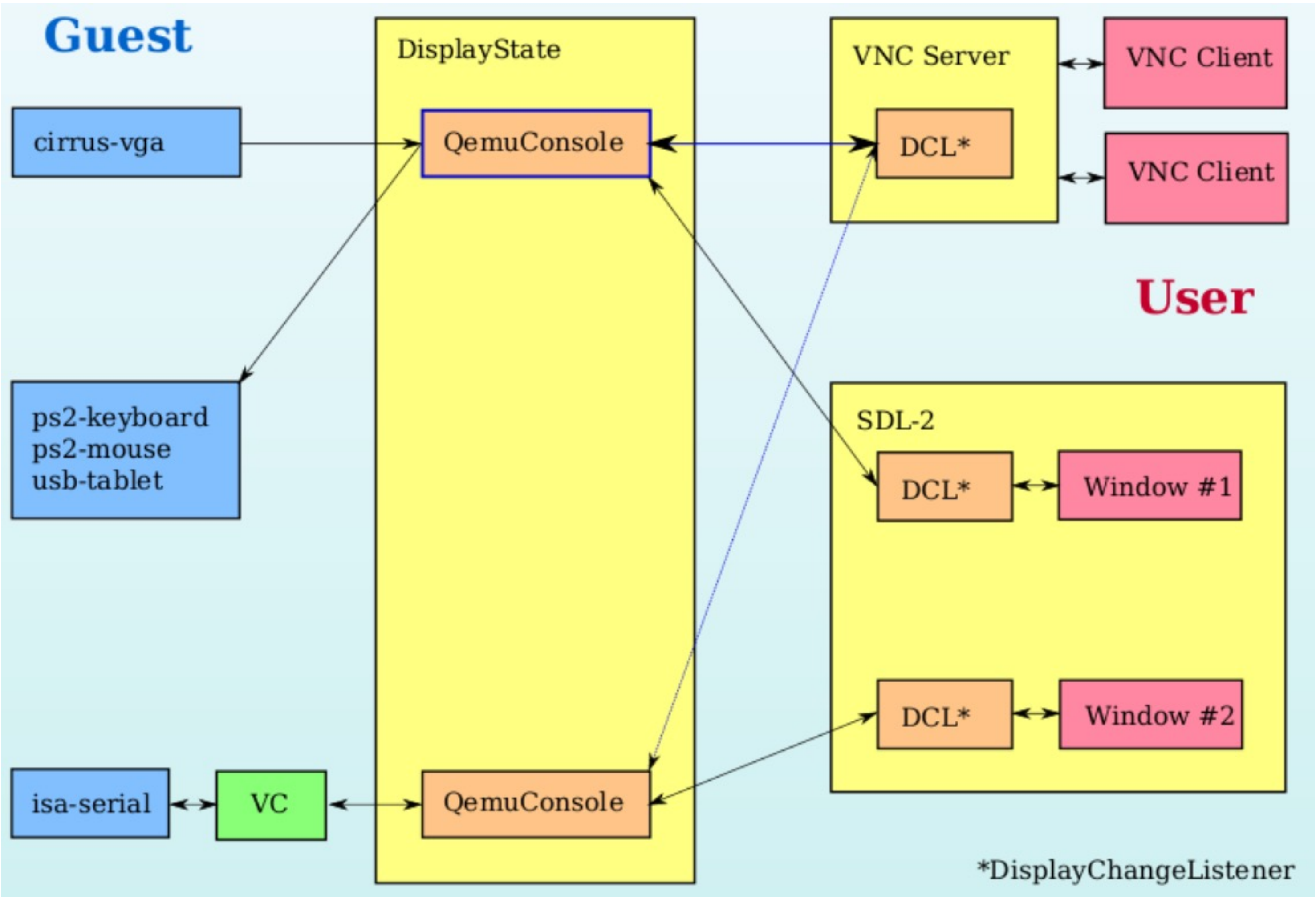
vga_init is to init vga-lowmem(0x90000-0xBFFFF) and register ioports callback function.

graphic_console_init hook up vga device with console core. *pci_std_vga_mmio_region_init* would init vga mmio region, and remap ioports into mmio region.

Another example: Virtual VGA

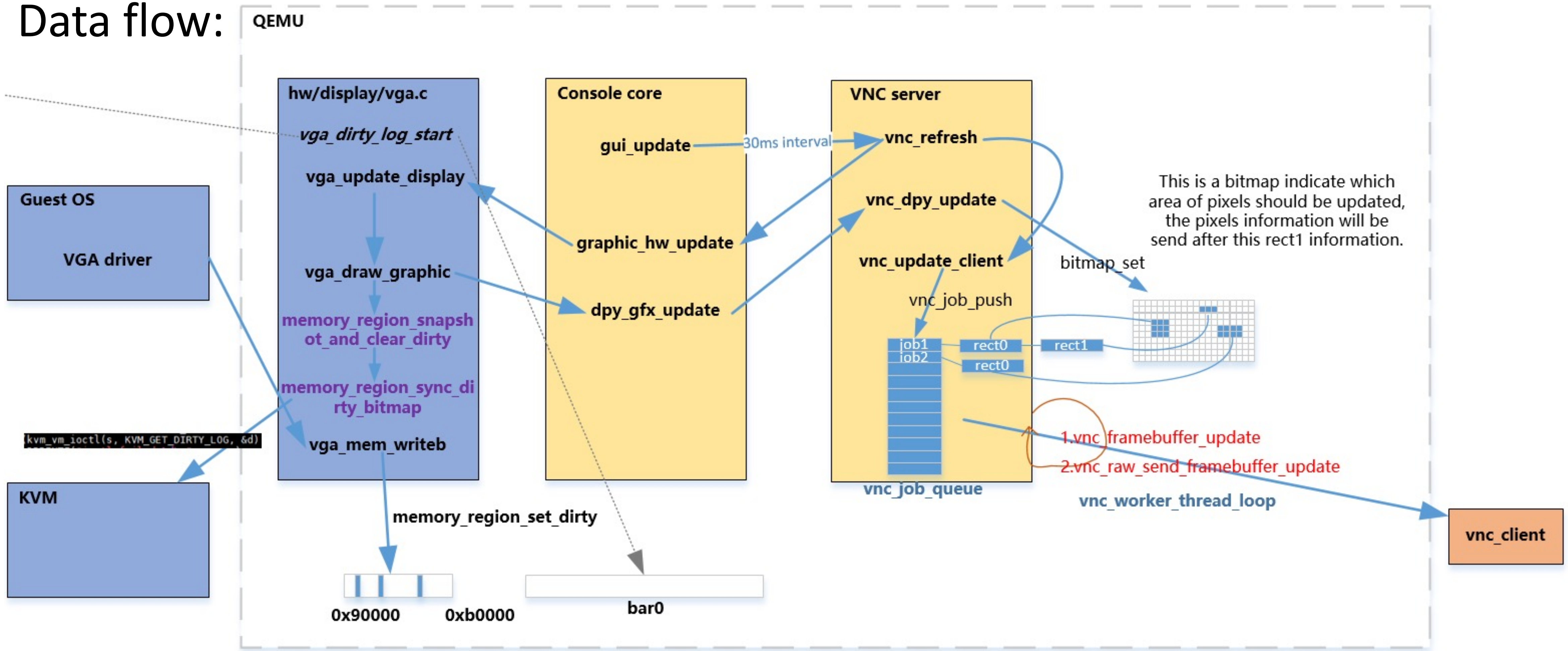
Big picture:

1. A DisplayState holds information about the machine.
2. A "console" is something that can write pixels into it’s DisplaySurface .
3. Blue box represents emulated hardware related to Display.
4. VNC is Virtual network computing.
5. DCL is mechanism to monitor DisplayState and inject user event to console.



Another example: Virtual VGA

Data flow:



Another example: Virtual VGA

Data flow:

1. gui_update is a timer callback function, will be execute every 30ms.
2. Then every listener's registed dpy_refresh will be called, such as vnc_refresh.
3. vnc_refresh call graphic_hw_update to update console surface.
4. vga_update_display then call vga_draw_graphic, get the dirty snapshot of the vram.
5. dpy_gfx_update dispatch the console state to all listener.
6. vnc_dpy_update mark dirty pixels in a bitmap according to vram dirty snapshot.
7. vnc_update_client wouch compose dirty rectangles from the bitmap, and push them into a vnc_jot_queue.
8. The worker thread pop the jobs in a loop, and sending the dirty rectangles info and it's pixel information to the vnc client.

Another example: Virtual VGA

How dirty log works:

Dirty loging mechanism is used in vga framebuffer update, code

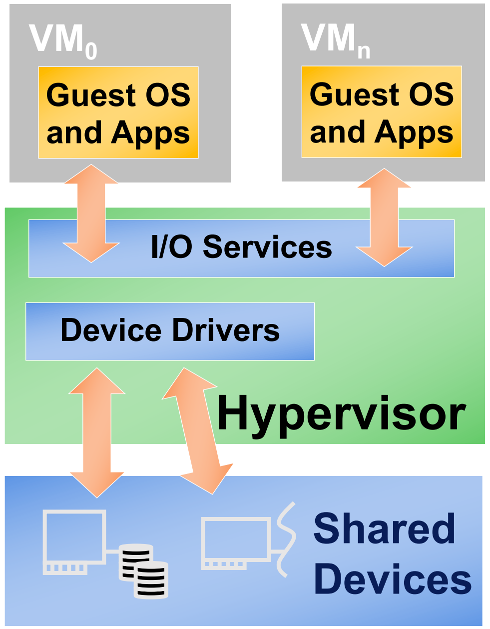
vga_dirty_log_start will enable log mechanism on vga vram memory region.

Loging mechanism is implemented by KVM assign special flag to kvm_userspace_memory_region.flag in __kvm_set_memory_region, kvm_userspace_memory_region is correspond to QEMU memory_region, then kvm_arch_commit_memory_region is called to map this piece memory and mark these pages are write-protected on the stage-2 MMU translation, thus, when guest write on these pages, VMExit occur, then kvm will parse the exit reason, kvm mark this page dirty.

QEMU could use an ioctl(KVM_GET_DIRTY_LOG)to collect these dirty information from KVM.

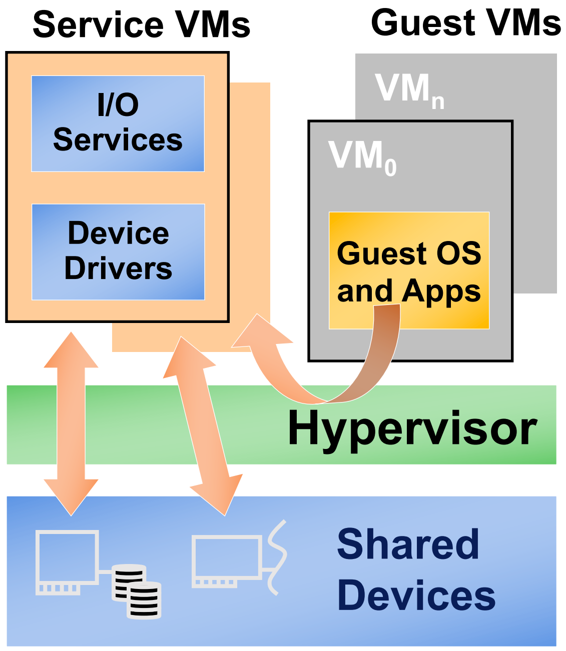
IO virtualization: VT-d

Monolithic Model



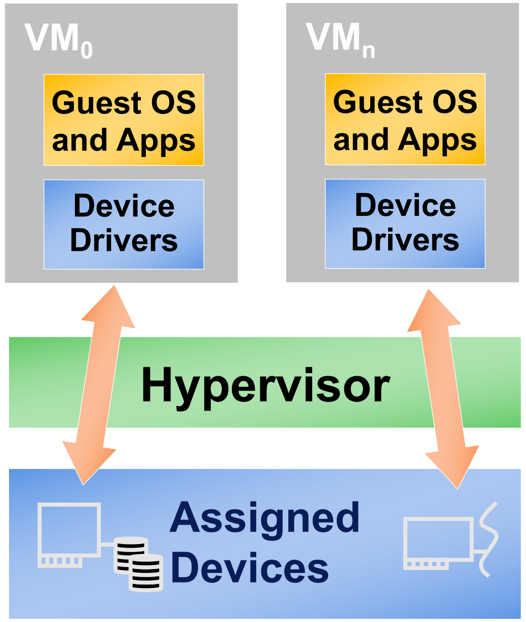
- Pro: Higher Performance
- Pro: I/O Device Sharing
- Pro: VM Migration
- Con: Larger Hypervisor

Service VM Model



- Pro: High Security
- Pro: I/O Device Sharing
- Pro: VM Migration
- Con: Lower Performance

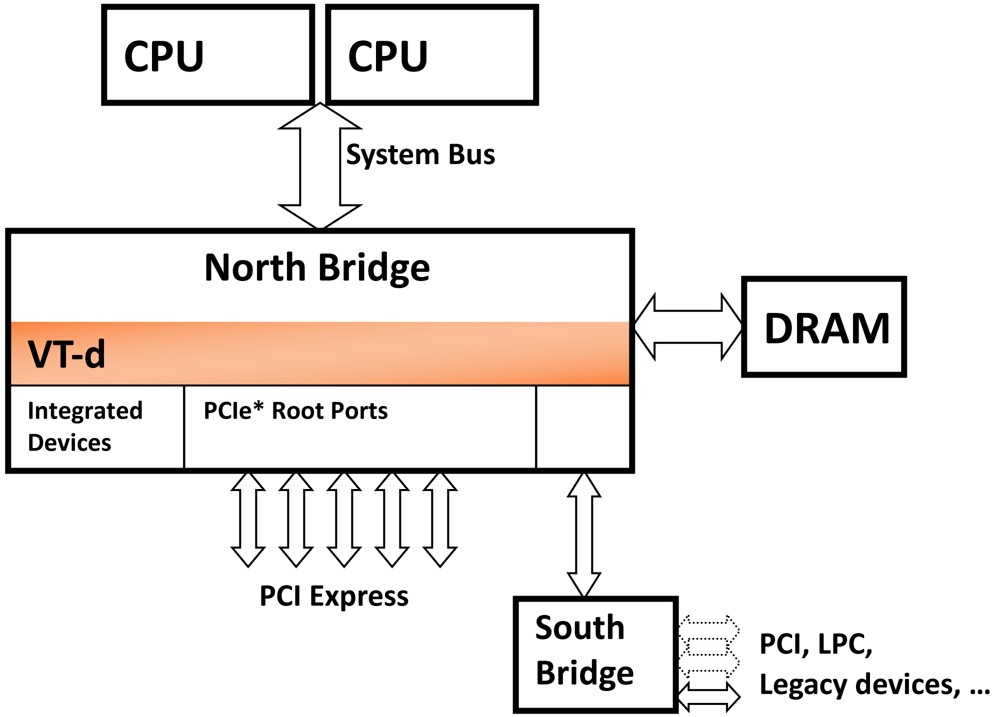
Pass-through Model



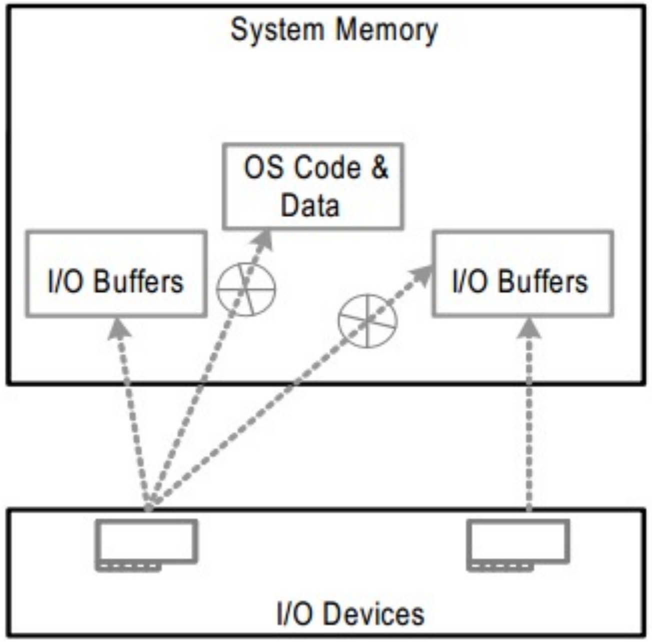
- Pro: Highest Performance
- Pro: Smaller Hypervisor
- Pro: Device assisted sharing
- Con: Migration Challenges

VT-d Overview

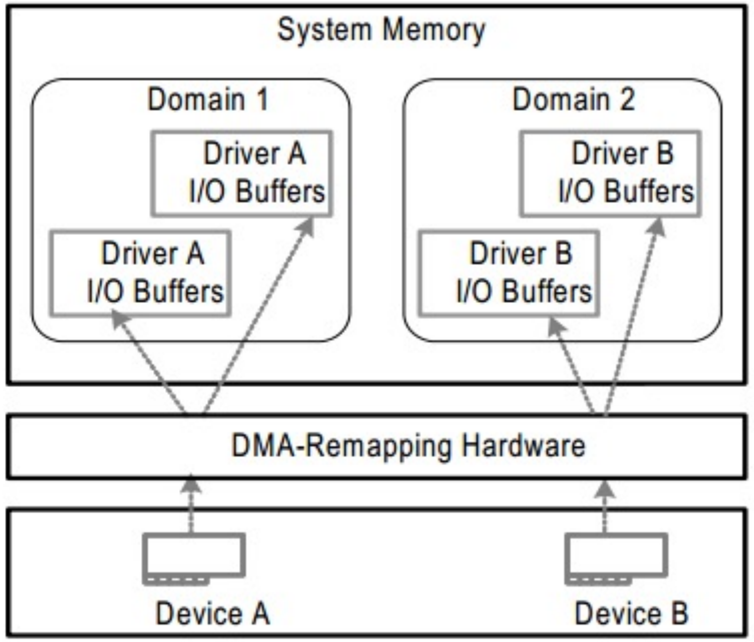
- VT-d is platform infrastructure for I/O virtualization
  - Defines architecture for DMA remapping
  - Implemented as part of platform core logic
  - Will be supported broadly in Intel server and client chipsets



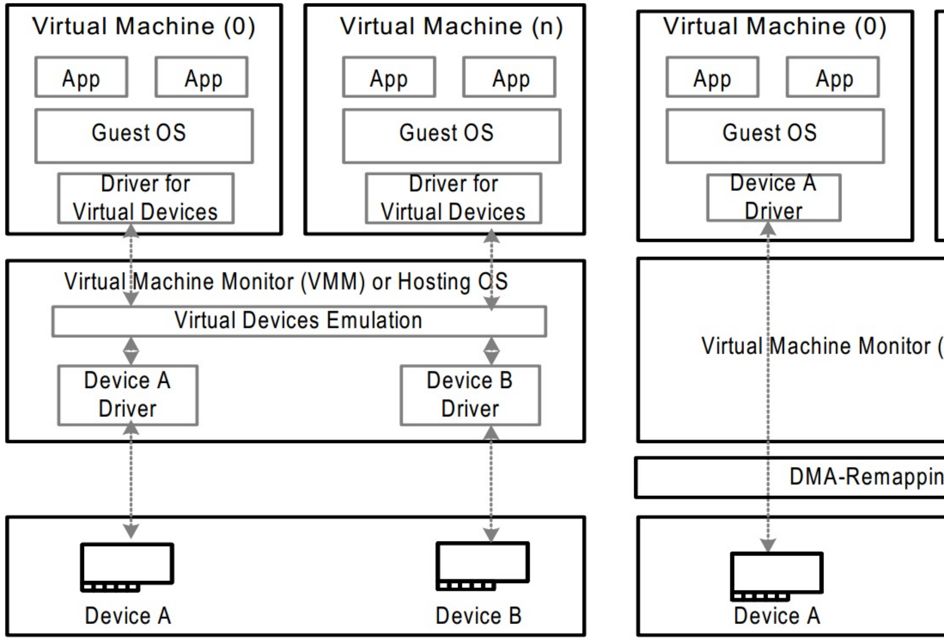
Add DMA remapping hardware component



Device DMA without isolation



Device DMA isolated using DMA remapping hardware



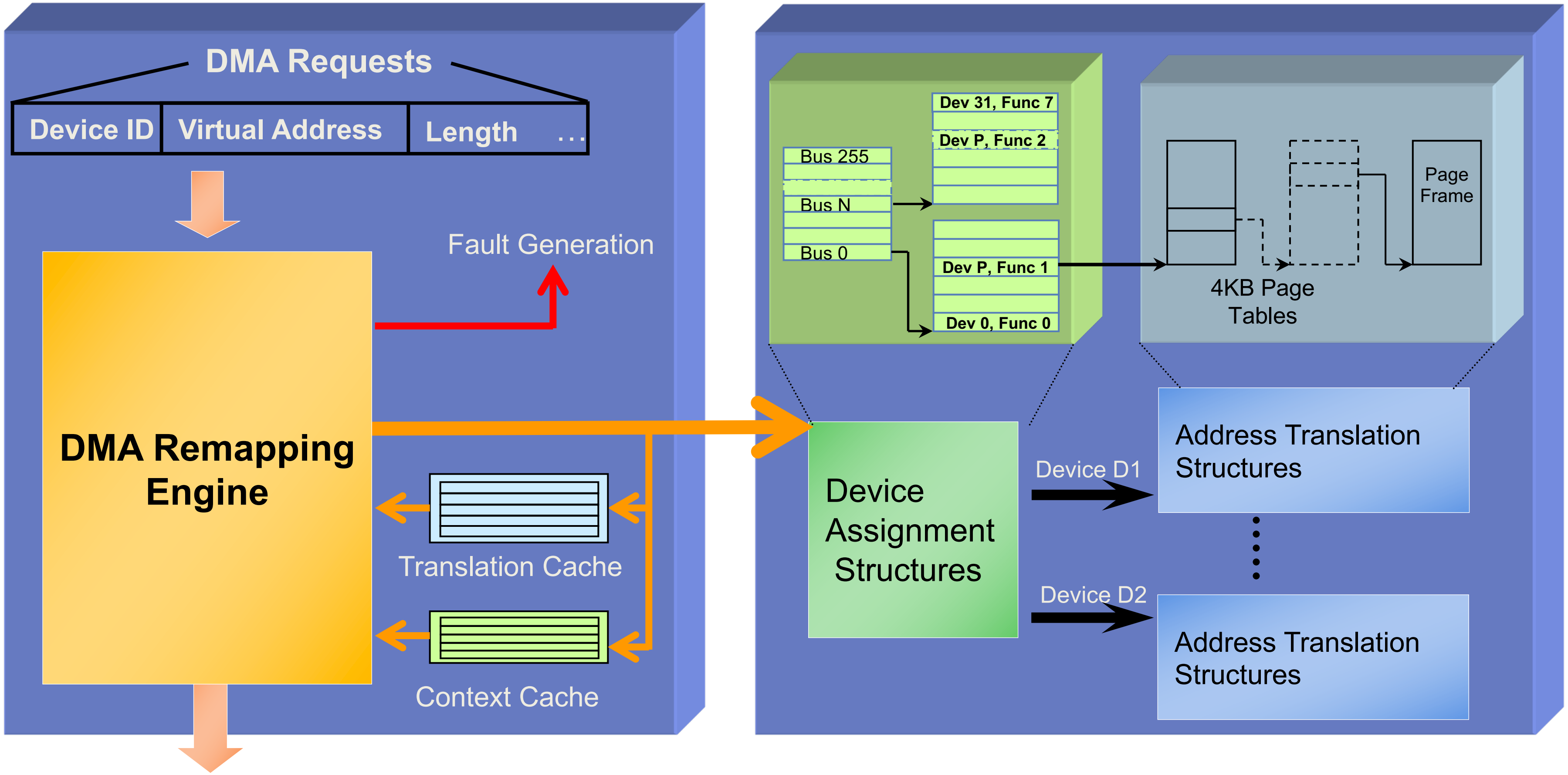
Example Software-based
I/O Virtualization

Direct Assignment of

Remapping Benefits

- Protection:
  - Enhance security and reliability through device isolation
  - End to end isolation from VM to devices
- Performance:
  - Allows I/O devices to be directly assigned to specific virtual machines
  - Eliminate Bounce buffer conditions with 32-bit devices
- Efficiency:
  - Interrupt isolation and load balancing
  - System scalability with extended xAPIC support
- Core platform infrastructure for Single Root IOV

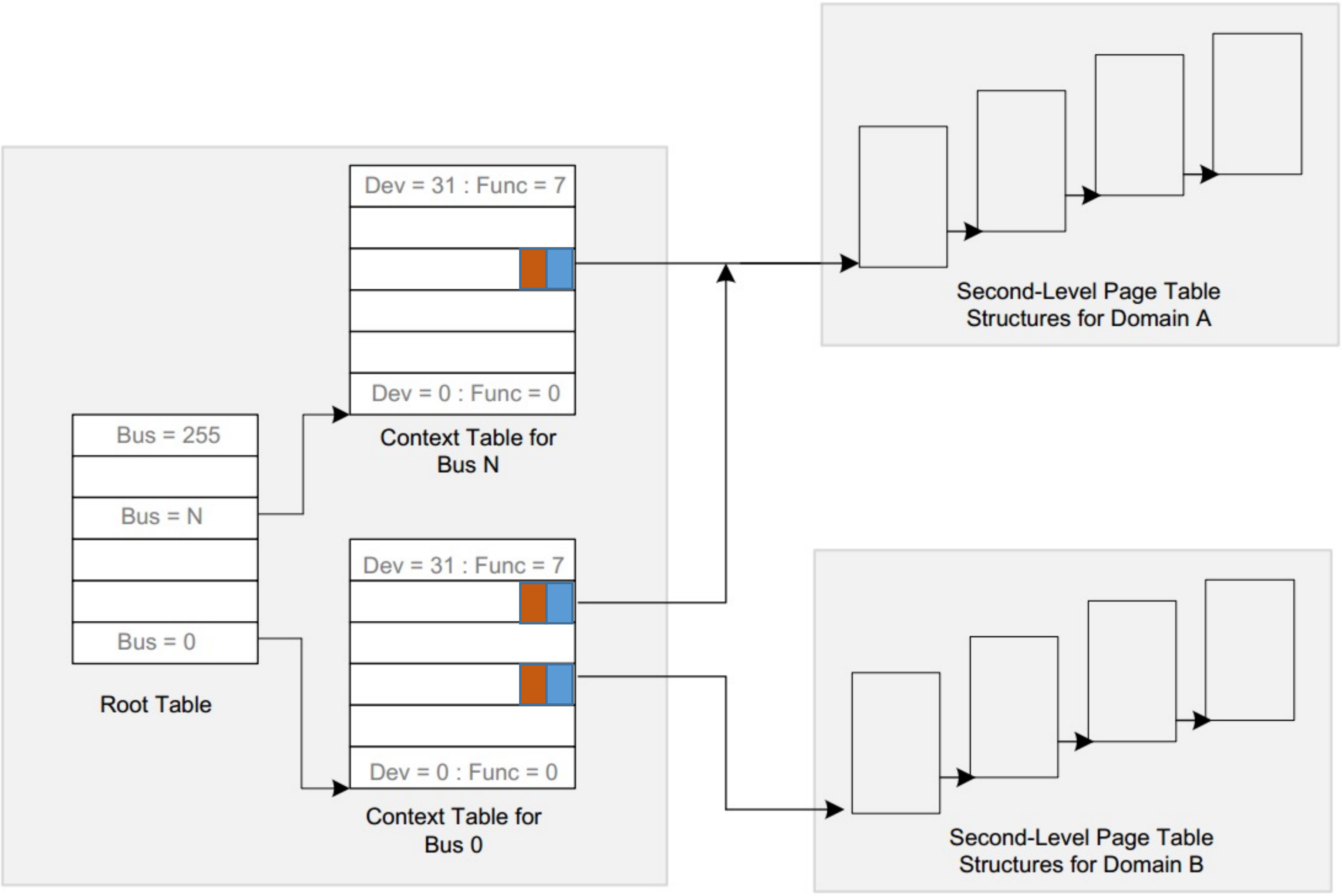
VT-d Architecture Detail(Need more…)

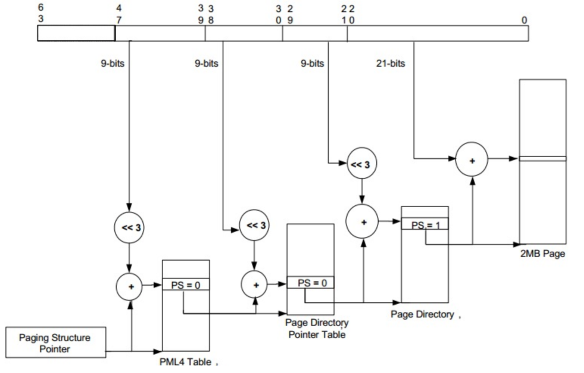


Memory Access with System Physical Address

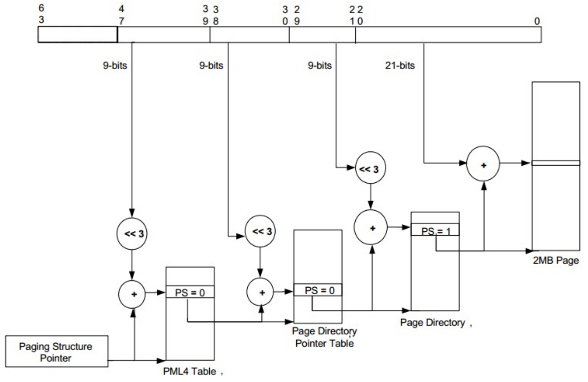
Memory-resident Partitioning And Translation Structures

VT-d: Hardware Page Walk Detail(Need more…)





Translate GPA to HPA



ASR-Address space root DID-domain ID

VT-x & VT-d Working Together