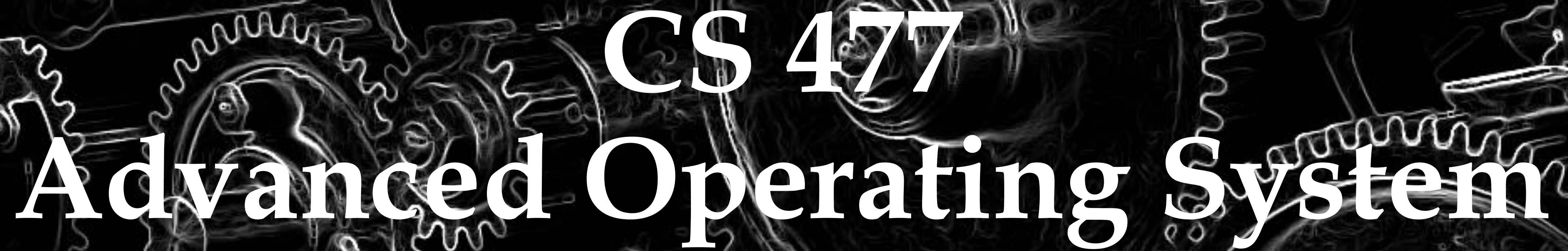


CS 477
Advanced Operating System

Lecture 14: Block IO & Virtualization

Today's agenda

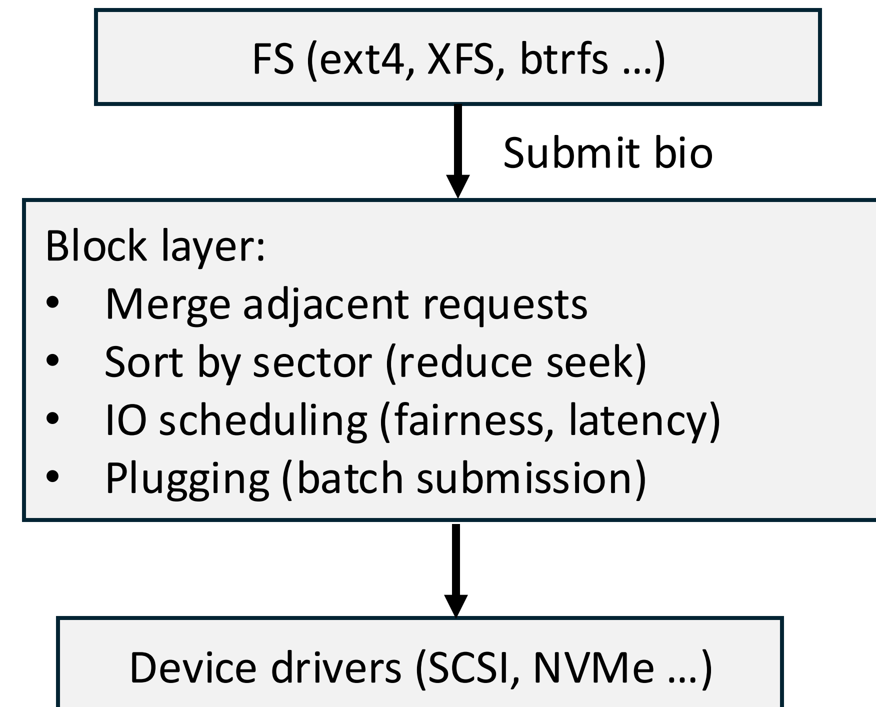
- **Block layer**
  - bio structure
  - Request scheduling
  - IO scheduler
- Virtualization

EPFL

Importance of block layer

- Without block layer:
  - Each FS talks directly to device drivers
  - Duplicate code for merging, sorting, scheduling
  - No optimization for disk seek patterns
- With block layer:

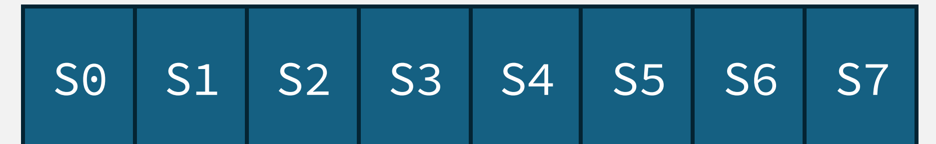
  Block layer virtualizes disk access like process scheduler virtualizes the CPU



EPFL

Anatomy of a block device

- Sectors (hardware view)
  - Minimum addressable unit of a device
  - Physical property → hard sector, device block
    - Typically 512 bytes (2KB for CD-ROM)
    - Modern drives: 4KB “advanced format”
- Block (software view)
  - Unit of file system IO
  - Multiple of sector size (device constraint)
  - Power of 2, <= page size (kernel constraint)
  - Typically 4KB or 8KB



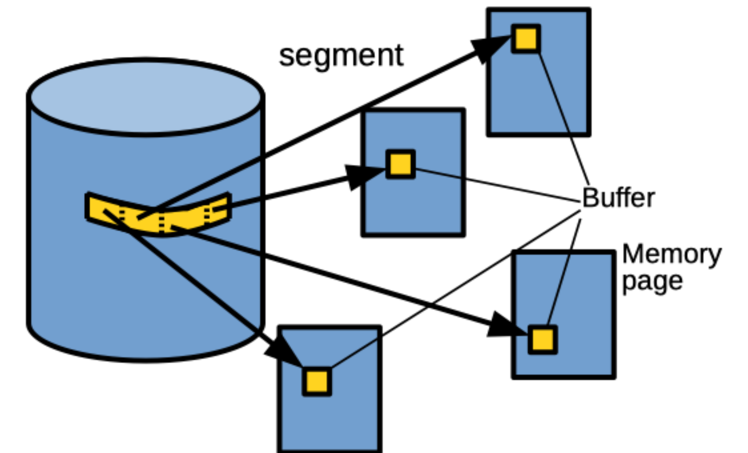
Kernel communicates in blocks, but the device driver speaks in sectors

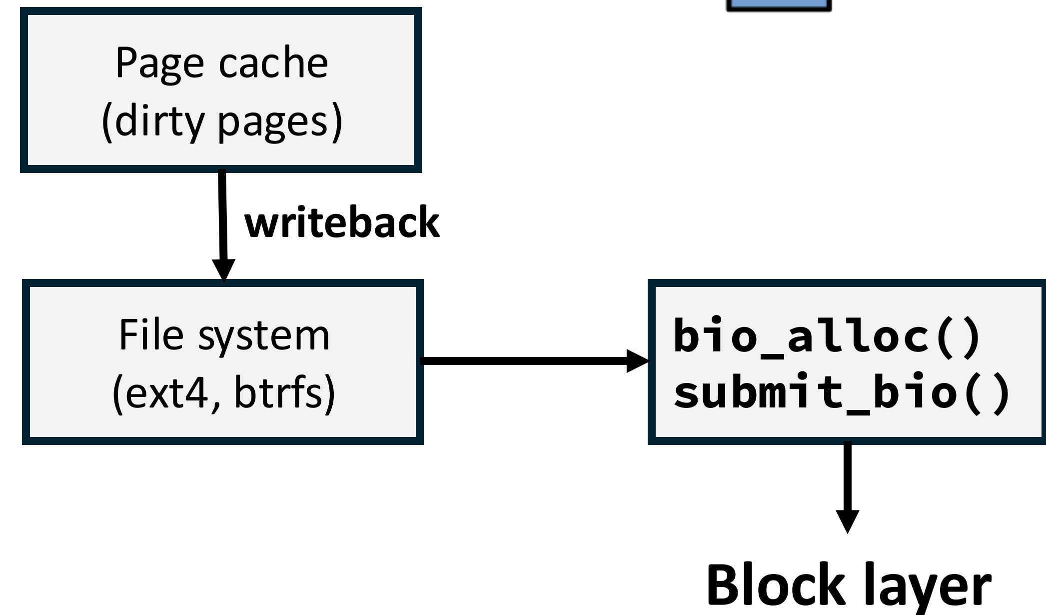
Q. Why do we care about this relationship?

EPFL

The `bio` structure

- Basic container for an active block IO operation
- A single IO can span multiple **non-contiguous memory pages**
- Scatter-gather support
- Light-weight, good for large IO





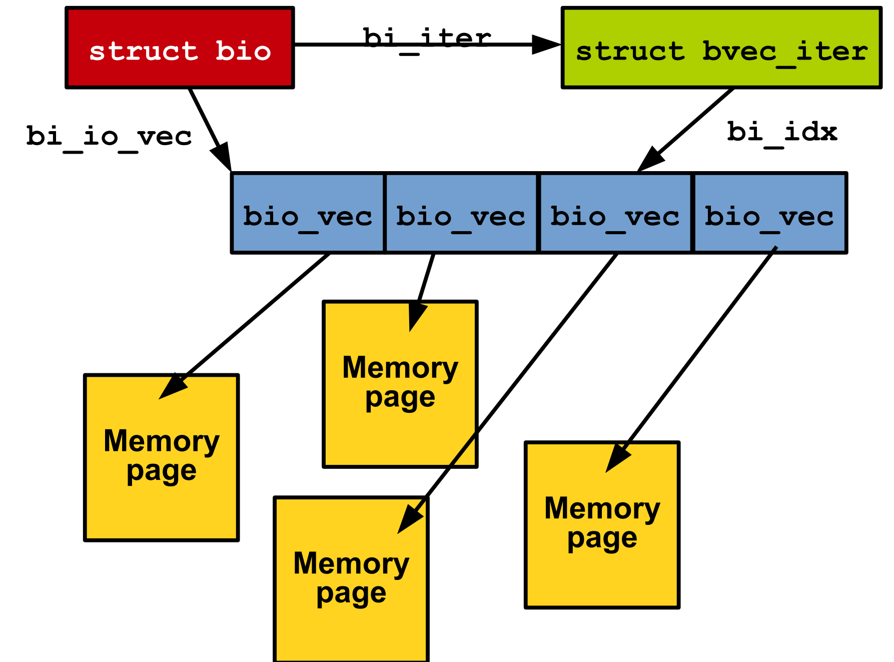
EPFL

bio struct

```
struct bio {
  struct bio *bi_next;                /* request list */
  struct block_device *bi_bdev;       /* target device */
  unsigned short bi_flags;            /* status flags */
  struct bvec_iter bi_iter;           /* position */
  bio_end_io_t *bi_end_io;            /* completion */
  void *bi_private;                   /* owner data */
  unsigned short bi_vcnt;             /* # of bio_vecs */
 struct bio_vec *bi_io_vec;           /* segment array */
};

struct bio_vec {
  struct page *bv_page;               /* physical page */
  unsigned int bv_len;                /* segment length */
  unsigned int bv_offset;             /* offset in page */
};

struct bvec_iter {
  sector_t bi_sector;                 /* target on disk */
  unsigned int bi_size;               /* remaining bytes */
  unsigned int bi_idx;                /* current bio_vec */
};
```

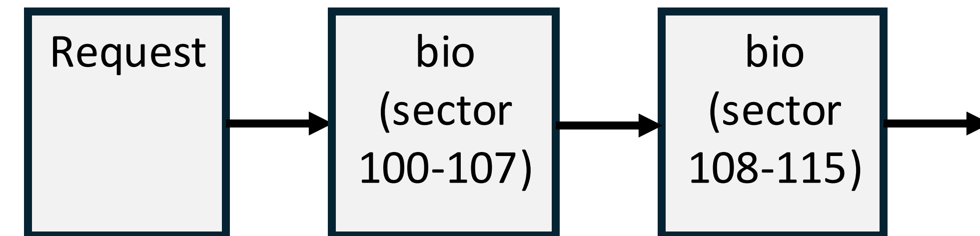


Satter-gather: single disk operation, multiple memory locations! DMA-friendly

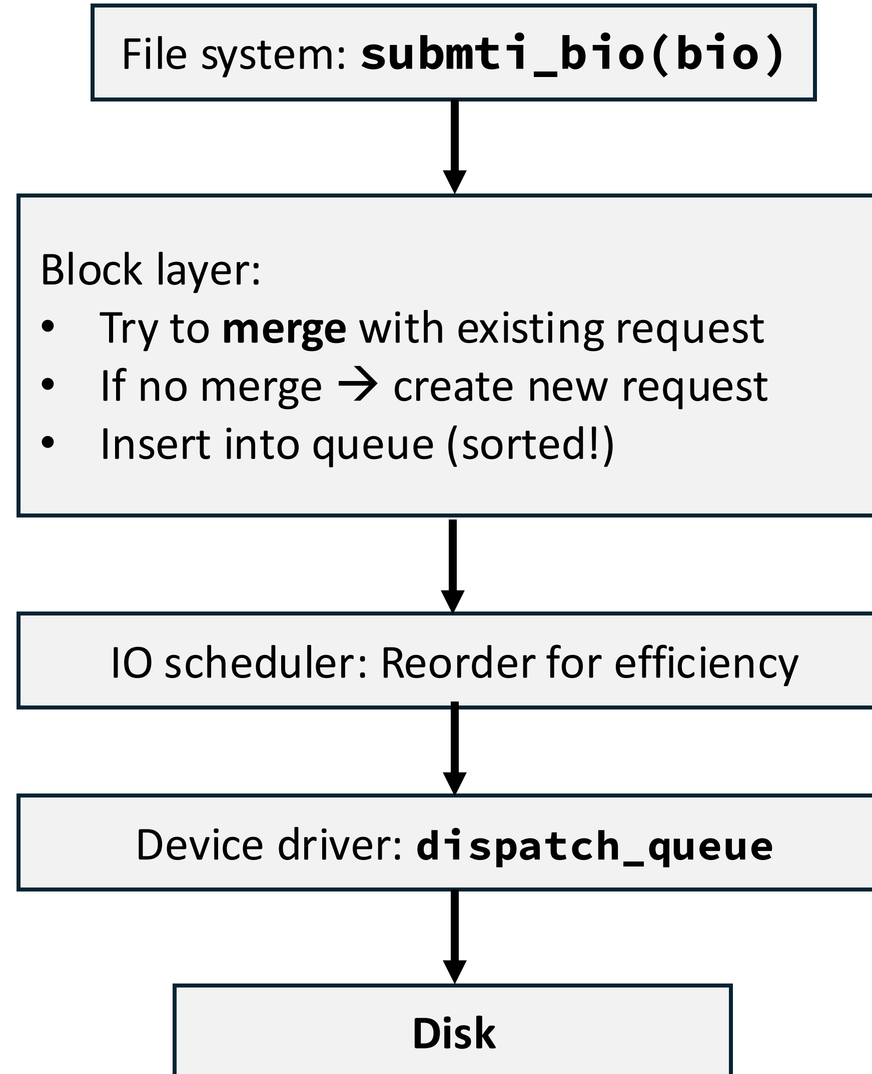
EPFL

Request queue

- Block devices maintain request queue for pending IO operations
- File system adds a request to the queue
- Block device driver removes the requests from the queue and submits to the device
- A single request:
  - Represented by struct request
  - Can operate on multiple consecutive disk blocks
    - Composed of one or more bio objects



Request flow



EPFL

IO scheduler: The issue

- Directly sending the requests to disk is sub-optimal
  - HDD seek time problem:
    - Random access (~10ms per seek) is 1000x slower than sequential (~0.01ms per block)
  - Example: Request order: 18, 5, 10, 1
    - Without scheduling: (|18 – 5|) + (|5 - 10|) + (|10 – 1|) = 27 sectors traveled
    - With sorting: 1, 5, 10, 18 ➔ 17 sectors traveled
    - Almost 40% reduction in head movement
  - Sorting algorithm is also known as the elevator algorithm
    - Define where an upcoming request should be added into the queue:
      - Front merge, back merge
      - Sorted insertion
    - Goal: minimize disk seek, best global throughput

Linux IO shedulers

- Noop/none: FIFO + merge only
  - Best for: SSD, NVMe, VM guests
- deadline: Sorted + deadline for starvation
  - Reads: 500 ms, writes: 5s
  - Best for: HDD, latency sensitive
- kyber: Simple 2-queue model
  - No seek time, best for NVMe SSDs

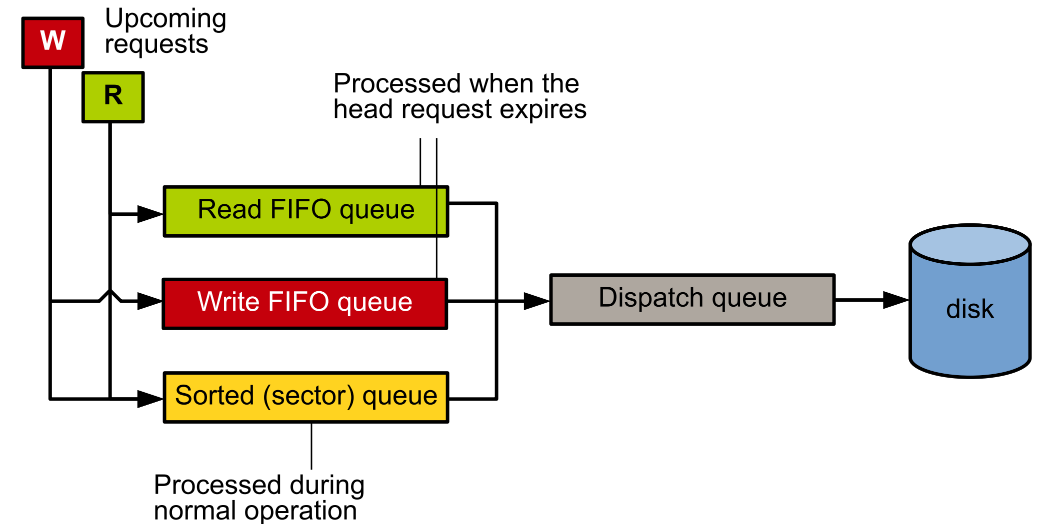
Tradeoff:

- HDD: Sort by sector → minimize seek
- SSD: No seek penalty → minimize overhead

EPFL

Deadline scheduler deep dive

- Problem with pure sorting: Writes can starve reads
  - Writes: buffered (async) → can wait
  - Reads: blocking (sync) → user waiting
- Solution: 3 queues + deadline
- Insight:
  - Normal case: Use sector-sorted order (good throughput)
  - Emergency case: Deadline expires → serve immediately (bounded latency)
  - Fairness: After some batches, allow writes to finish



EPFL

Simplified deadline scheduler algorithm

- On request arrival:
  - Set a deadline for a specific request type (read: 0.5 s, write: 5s)
  - Add to FIFO queue (by arrival time)
  - Add to sorted tree (by sector number)
- On dispatch:
  - Priority 1: serve the oldest expired reads
  - Priroity 2: serve the oldest expired writes
  - Priority 3: Normal operation
    - If reads are available and writes are not starved, then serve reads
      - Example: serve 16 reads before serving a write
    - Else, serve the next write by sector order

Today's agenda

- Block layer
  - bio structure
  - Request scheduling
  - IO scheduler
- **Virtualization**
  - Processor virtualization
  - Memory virtualization
  - IO virtualization

What is virtualization?

Virtualization enables running multiple OSes and applications on the same physical hardware simultaneously

- **Resource multiplexing:** Share physical resources among multiple guests
- **Isolation:** Each guest operates independently, unaware of others
- **Abstraction:** Present uniform virtual hardware regardless of underlying physical hardware

Insight: Illusion that each guest OS has exclusive access to the hardware when in reality they're sharing it

Virtual machine monitor (VMM)

VMM is a software layer:

- Manages access to physical resources
- Provides isolation between VMs
- Creates and presents virtual hardware to guest OSes
- Intercepts and handles privileged operations

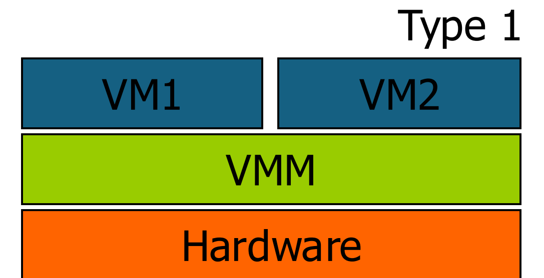
VMM ←→ Hypervisor, host

VM ←→ Guest OS

Guest ←→ Virtual machine (VM)

Type 1 Hypervisor (Bare-metal)

- Hypervisor runs directly on hardware
  - No host OS between hypervisor and hardware
  - Direct hardware access → better performance
  - Smaller attack surface

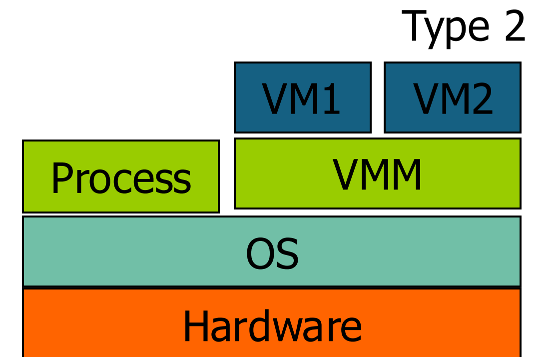


- Example: Xen, Vmware ESX, Microsoft Hyper-V
- Use case: Server virtualization, cloud infrastructure

EPFL

Type 2 Hypervisor (Hosted)

- Hypervisor runs as a process on the host OS
  - Host OS manages hardware
  - Easier to install and use
  - Can coexist with other applications



- Example: KVM, Vmware Workstation, VirtualBox
- Use case: Server virtualization, Desktop virtualization development, testing, cloud infrastructure

Why virtualization matters

- Isolation
  - Security breaches in one VM don't affect others
  - Reduces the spread of risks across workloads
- Rollback
  - Snapshots allow quick recovery
  - Quickly recover from security breaches
- Abstraction
  - VMs decoupled from specific hardware
  - Limits direct access to hardware
- Efficiency
  - Better hardware utilization through consolidation
  - "Green IT": fewer physical servers needed

EPFL

Application domains

- Server virtualization
  - Data center consolidation, cloud infrastructure
  - KVM, Xen, VMware ESX server
- Desktop virtualization
  - Run multiple OSes on one machine
  - VMware, VirtualBox, Citrix Xen HDX
- Mobile virtualization
  - Secure execution environment
  - Xen on ARM, KVM
- Cloud computing
  - Storage/platform cloud services
  - Amazon EC2, Microsoft Azure, Google CE
- Emulation
  - iPhone/Android emulator
  - Qemu, Bochs

Today's agenda

- Block layer
  - bio structure
  - Request scheduling
  - IO scheduler
- Virtualization
  - **Processor virtualization**
  - Memory virtualization
  - IO virtualization

EPFL

The fundamental challenge

"For an architecture to be classically virtualizable, **all sensitive instructions must be privileged instructions." –** Popek & Goldbeg (1974)

- **Sensitive instruction:**
  - Instructions that change process mode (eg, modifying privilege levels)
  - Instructions that access hardware directly
  - Instruction whose behavior differs between user/kernel modes
- **The x86 problem:**
  - Some sensitive instructions are **not** privileged
  - They don't trap when executed in user mode
  - Instead, they silently fail or return different results

Breaks the classic trap-and-emulate model!

EPFL

The x86 problem

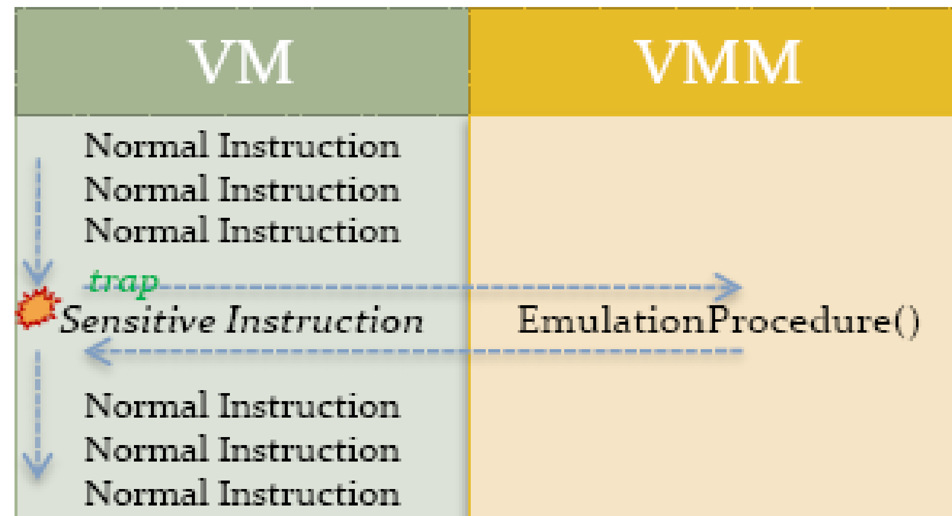
- Some instructions (**sensitive**) read or update the state of VM and don't trap (**non-privileged**)
  - 17 sensitive, non-privileged instructions

Group	Instructions
Access to interrupt flag	**`pushf, popf, iret`**
Segment manipulation instructions	**`pop<seg>, push<seg>, mov<seg>`**
Read-only access to privileged instructions	**`sgdt,`**`sldt, sidt, smsw`
Interrupt and gate instructions	`fcall, longjump, retfar, str,` **`int<n>`**

- **`popf`** doesn't update the interrupt flag (**`IF`**)
  - Impossible to detect when the guest disables interrupts
- **`push %cs`** can read code segment selector (%cs) and learn its CPL
  - Guest gets confused

Classic virtualization: Trap & Emulate

- Guest runs in user mode (Ring 3)
- Privileged instructions trap to VMM
- VMM emulates the instruction



Four approches to CPU virtualization

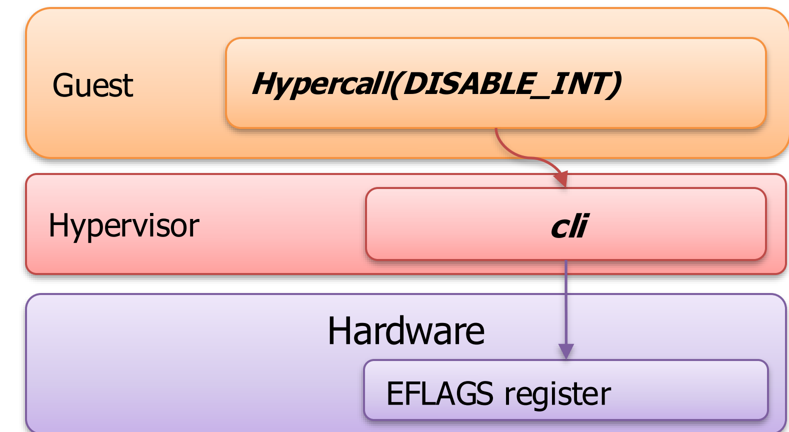
1. Para-virtualization: Modify guest OS to avoid problematic instructions
2. Emulation: Impelment entire CPU in software
3. Binary translation: Rewrite problematic instructions at runtime
4. Hardware-assisted: CPU extensions for native virtualization

EPFL

Approach 1: Para-virtualization

Key idea: modify the guest OS to be "virtualization-aware"

- Guest explicitly calls the hypervisor using **hypercalls** to execute problematic instructions
  - cli → guest OS executes on receiving hypercall(DISABLE_INT)
- Pros:
  - Near-native performance
  - No hardware support required
  - Clan interface between guest and hypervisor
- Cons:
  - Requires guest kernel modification
  - Cannot run unmodified proprietary OSes
  - Maintenance burden: must port to each OS



Example: Xen, KVM

Approach 2: Emulation

- Implement the entire CPU in software
- CPU emulation:
  - Fetches and decodes the next instruction
  - Executes using the emulated registers and memory
- Pros:
  - No hardware support required
  - Simple conceptually
  - Great for debugging and testing

Cons:

- Very slow
- Each instruction has a huge overhead

Example: Bochs

```
addl %ebx, %eax
```

```
unsigned long regs[8];
regs[EAX] += regs[EBX];
```

EPFL

Approach 3: Binary translation

- Translate problematic instructions at runtime (JIT)
- Process:
  1. Run safe instructions directly on hw
  2. Detect and rewrite safety instructions
  3. Cache translated code blocks for reuse
- Pros:
  - No hardware support required
  - Good performance (near-native for most)
- Cons
  - Complex implementation (x86-to-x86 JIT compiler)

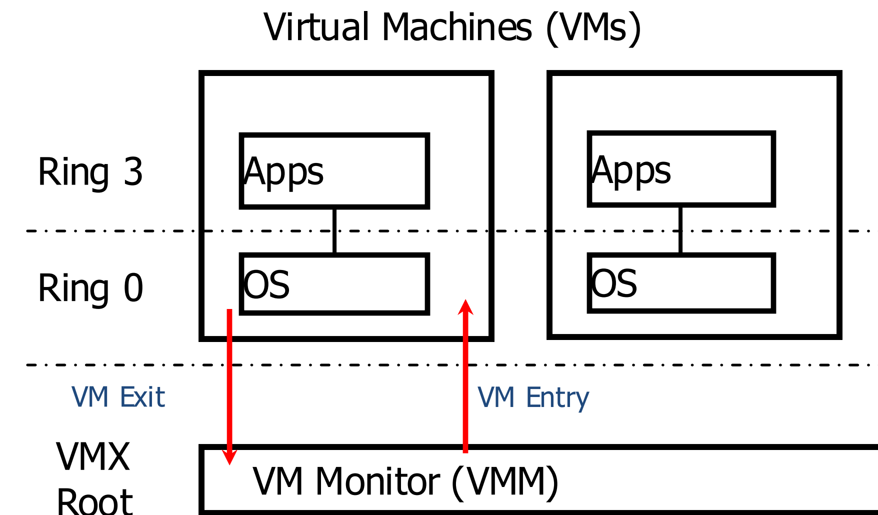
Original code	Translated code
`mov %eax, %cr3`	`call vmm_handle_cr3_write`
`cli`	`call vmm_disable_inetrrupts`
`add %ebx, %eax`	`add %ebx, %eax`
`ret`	`jmp translation_cache_lookup`

Solutions: VMware Workstation (for pre-HW support), Qemu

EPFL

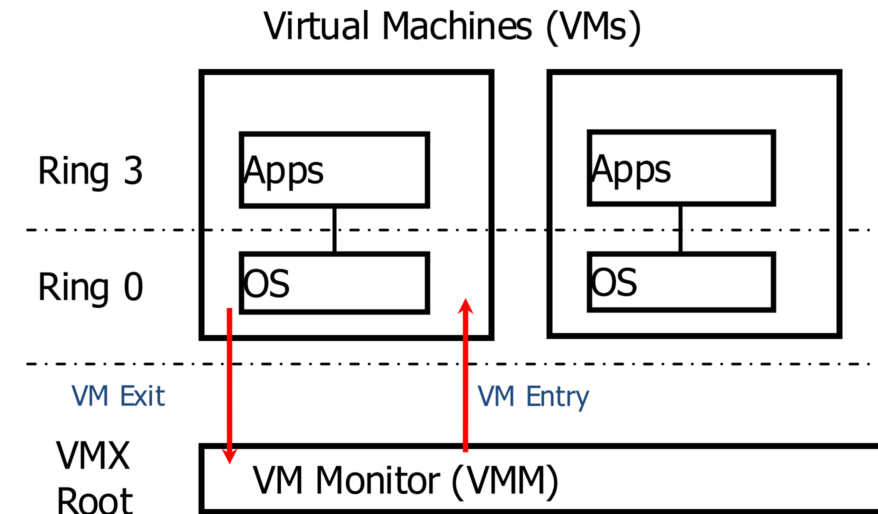
Approach 4: Hardware-assisted virtualization

- Extend the CPU with native virtualization support
- Intel VT-x/AMD SVM add new processor mode:
- New execution mode + new instruction set and control structure
  - Guest OS runs at Ring 0, but in a “**less privileged**” VMX non-root mode



Intel VMX architecture

- **Two new operating modes:**
  - VMX non-root: Less privileged, for guest execution
  - VMX root: More privilged, for VMM execution
- **Two new transitions:**
  - VM entry: VMM → guest
  - VM exit: guest → VMM
- **VM control structure (VMCS):**
  - Stores guest and host CPU state
  - Controls which operations cause VM exits
  - Loaded/stored on transitions
- **Execution controls determine when exits occur:**
  - Access to privileged state (CR3, etc.)
  - Occurrence of exceptions
  - IO operations



Hardware-assisted: pros and cons

- Advantages
  - Fast (near-native performance)
  - Runs completely unmodified guests
  - Clean architectural support
  - Simpler VMM implementation
- Disadvantages:
  - Requires specific hardware (Intel VT-x, AMD SVM)
  - VM exits still have overhead (virtualization tax)

Example: KV, modern Xen, VMware (current)

Today's agenda

- Block layer
  - bio structure
  - Request scheduling
  - IO scheduler
- Virtualization
  - Processor virtualization
  - **Memory virtualization**
  - IO virtualization

The address space challenge

- **Multiple levels of address spaces**

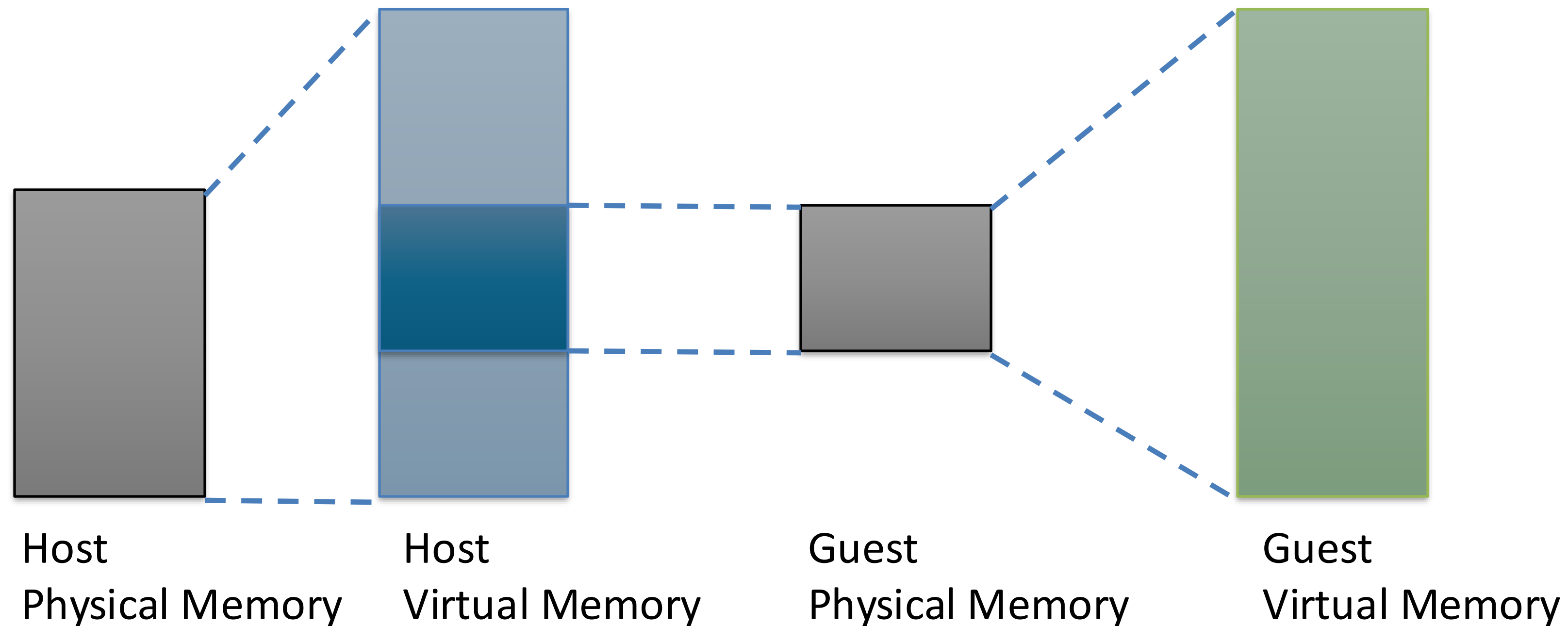
```
Guest virtual address (GVA)
      ↓ (guest page tables)
Guest physical address (GPA) ← What guest OS "thinks" is physical
      ↓ (???)
Host physical address (HPA) ← Actual machine memory
```

- **Problem:**
  - Guest OS manages its own page tables (GVA → GPA)
  - Hardware MMU needs mappings to real physical addresses

Q. How do we translate GVA → HPA?

EPFL

Memory virtualization: visual overview



- **Insight:** Guest's "physical" memory is actually virtual from the host's perspective

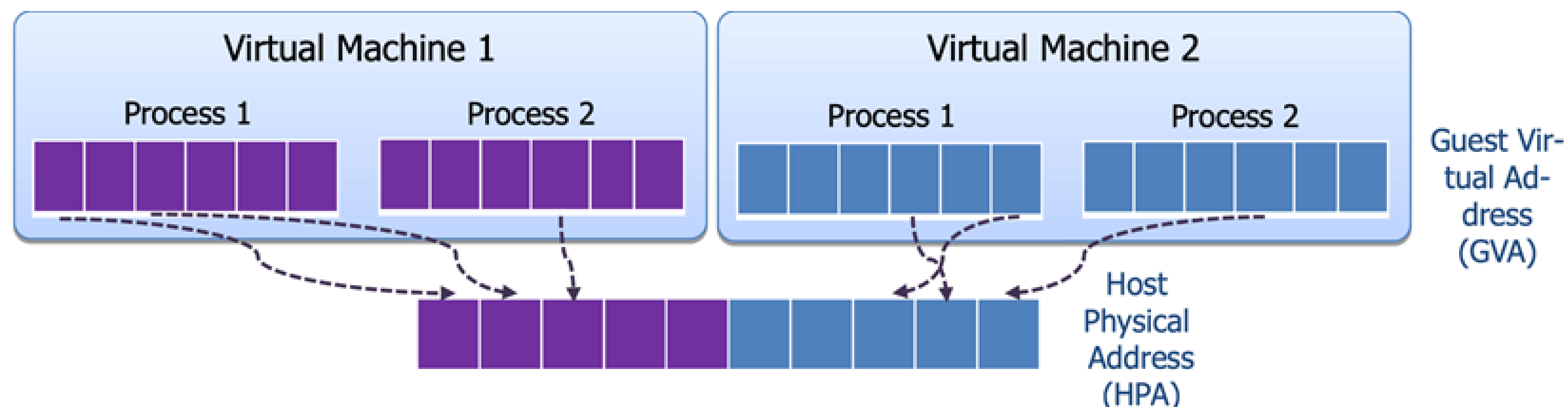
EPFL

Three approaches to memory virtualization

Approach	Key idea	Guest modification?	HW support?
Direct paging	Guest directly uses HPAs	Yes	No
Shadow paging	VMM maintains hidden page tables	No	No
Nested paging	Hardware walks two page tables	No	Yes

Approach 1: Direct paging

- Guest OS directly maintains GVA → HPA mappings
- Hypervisor allocates fixed physical memory region to each guest at boot
- Guest kernel modified to use HPAs directly in page tables



Direct paging: pros and cons

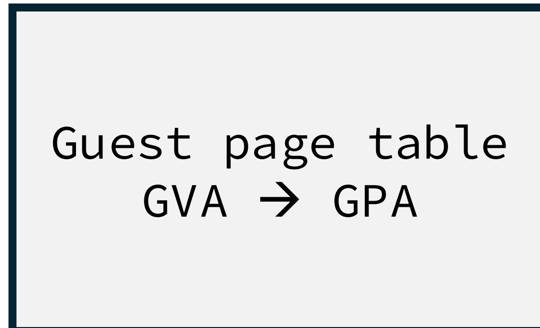
- Advantages
  - Simple to implement
  - No virtualization overhead on memory access
  - Higher performance possible
- Disadvantages:
  - Requires guest kernel modification (para-virtualization)
  - Inflexible: memory allocation fixed at VM creation
  - Cannot run closed-source guest OSes
  - No memory overcommitment

EPFL

Approach 2: Shadow paging

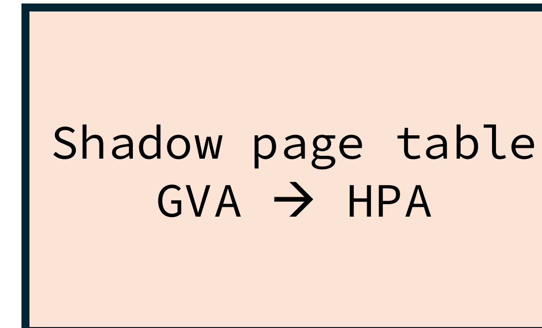
VMM maintains hidden **shadow** page tables that hardware uses

Guest's view



VMM keeps in sync →

Reality (what HW uses)



Guest thinks these are good

MMU actually uses it

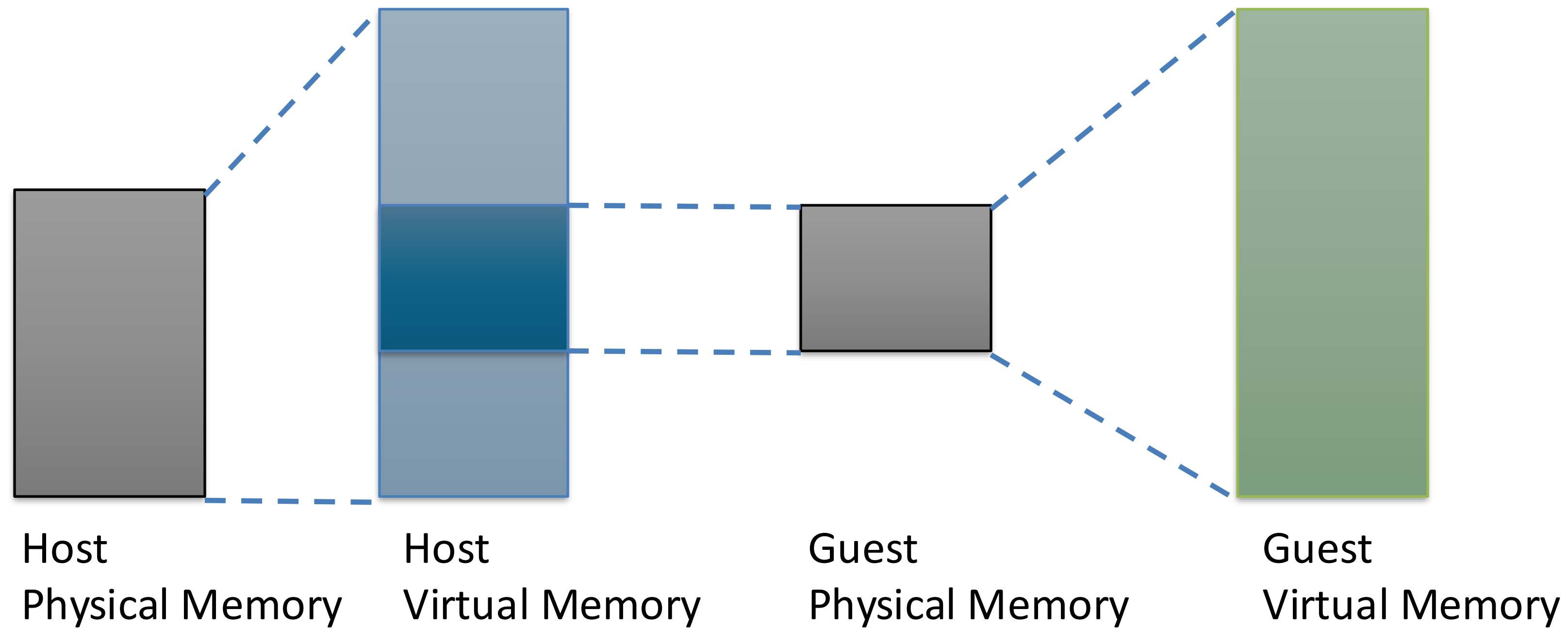
EPFL

Shadow paging working

- Process:
  1. Guest OS maintains its own page tables (GVA → GPA) as normal
  2. VMM intercepts all guest page table modifications
  3. VMM creates corresponding shadow page tables (GVA → HPA)
  4. Hardware MMU uses shadow page tables, not guest tables
  5. VMM keeps shadow tables synchronized with guest tables
- Synchronization mechanism:
  - Write-protect guest page table pages
  - Every guest PTE modification causes a trap to VMM
  - VMM updates shadow table and returns

EPFL

Shadow paging memory layout



MMU uses shadow PT from VMM

One-to-one mapping via host kernel

Guest's "physical" is virtual

Guest page table (**not** used by MMU)

Shadow paging: pros and cons

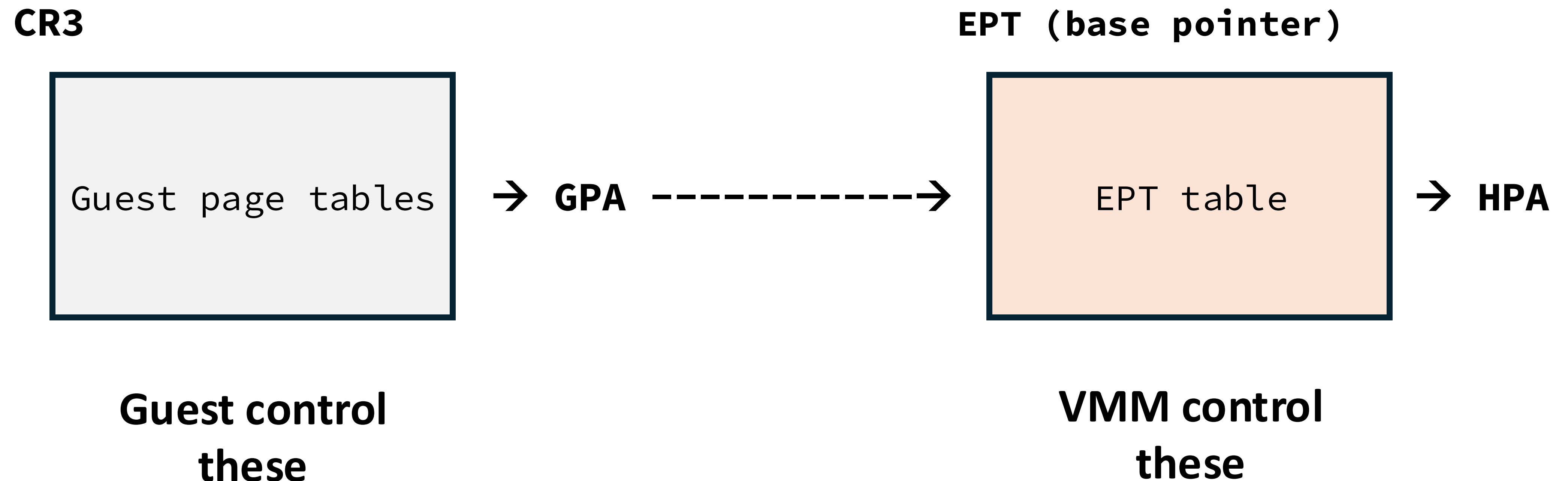
- Advantages
  - Supports unmodified guest OSes
  - No hardware extension required
  - Allows memory overcommitment
- Disadvantages
  - Complex implementation
  - High overhead: every page table modification traps
  - Memory overhead: Shadow tables for each address space
  - TLB flush on context within guest

Example: Still used in nested virtualization case

EPFL

Approach 3: Nested paging (hardware-assisted)

- Idea: Let hardware perform both levels of translation
- Intel EPT (extended page table) / AMD NPT (nested page table)



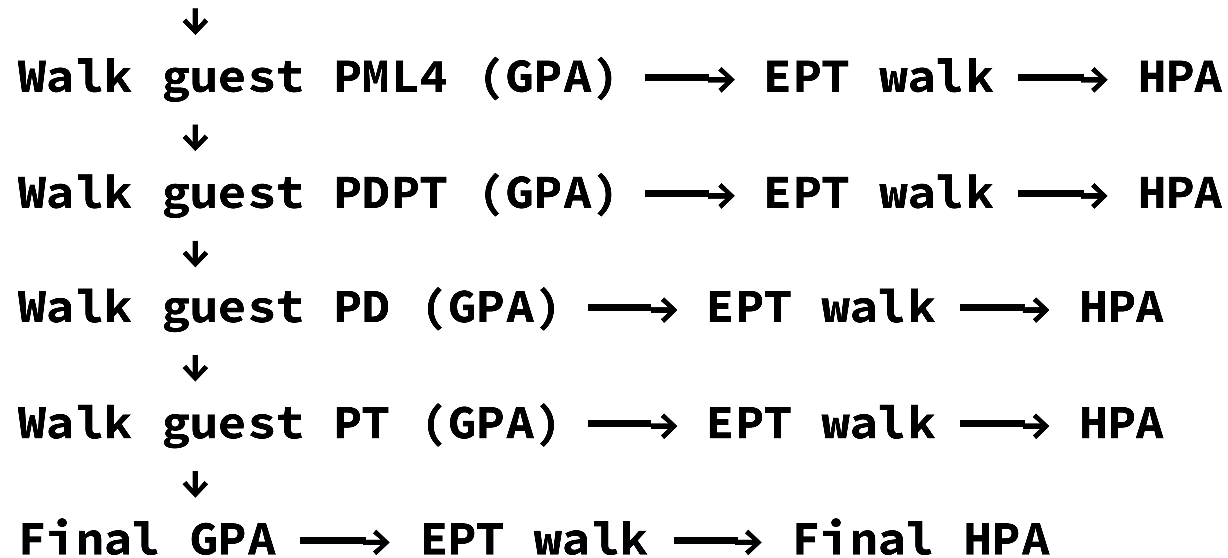
EPFL

Nested page table working

- On every memory access:
    1. Guest maintains its page tables (GVA → GPA) normally
    2. VMM maintains EPT/NPT (GPA → HPA)
    3. CPU walks guest tables to get GPA
    4. For **each GPA encountered**, CPU walks **EPT** to get **HPA**
    5. Final translation: GVA → GPA → HPA
- EPT activation:
    - EPT base pointer loaded from VMCS on VM entry
    - PET active only in VMX non-root mode
    - Deactivated on VM exit

EPT translation: detailed walkthrough

- For a 4-level page table with EPT:



- Worst case: Up to **24 memory accesses** per TLB miss

EPFL

EPT violations

- EPT also specifies access permissions:
  - Read/write/execute permissions per page
  - Supervisor vs. user access
- EPT violation: Attempt to access memory in an unauthorized way
  - Causes VM exit to hypervisor
  - VMM can handle (eg, CoW, lazy allocation)
- Use cases:
  - Memory overcommitment
  - Live migration dirty page tracking
  - Memory deduplication (KSM)

EPFL

Nested page table: pros and cons

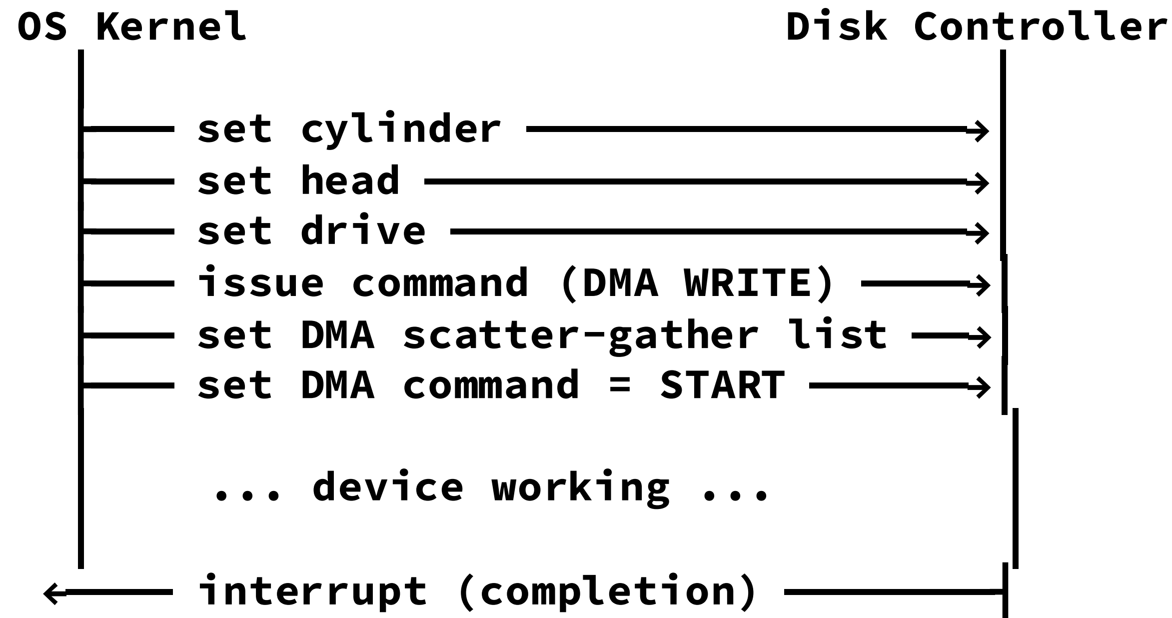
- Advantages
  - Simple VMM implementation (no shadow table maintenance)
  - Supports unmodified guest OSes
  - Low overhead for page table modifications
  - No trap on guest page table changes
- Disadvantages
  - Requires hardware support
  - Larger TLB footprint needed
  - Page walk amplification on TLB miss
  - TLB miss more expensive than shadow paging

Today's agenda

- Block layer
  - bio structure
  - Request scheduling
  - IO scheduler
- Virtualization
  - Processor virtualization
  - Memory virtualization
  - **IO virtualization**

OS and device interaction

- Simplified disk IO operation:



Challenge: How do we virtualize this for multiple VMs?

Three approaches to IO virtualization

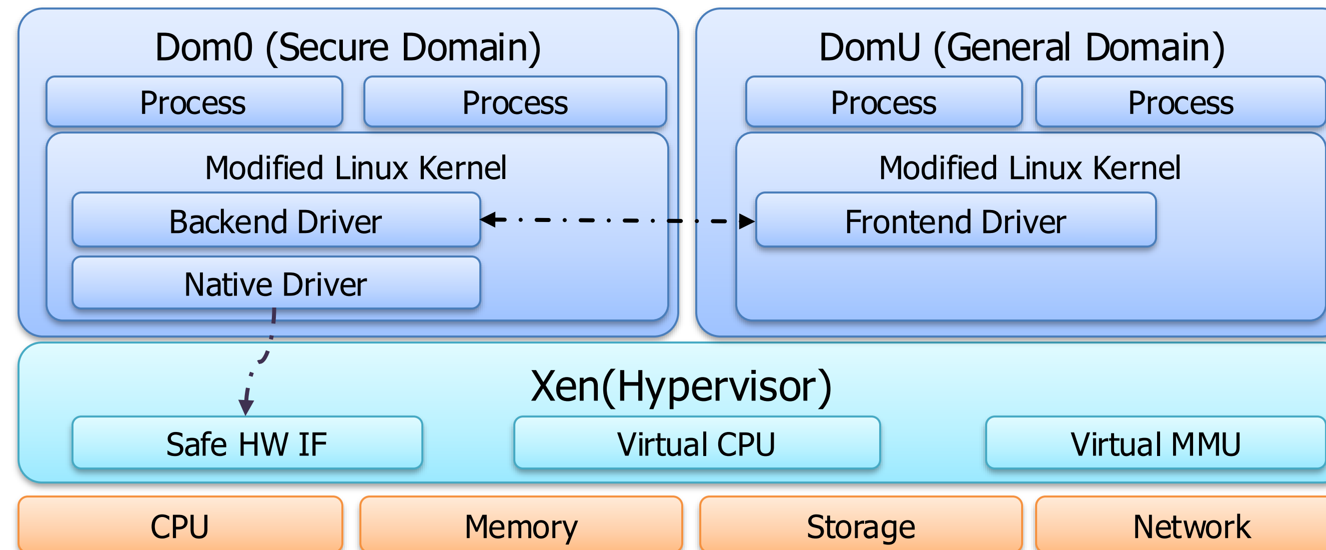
1. Front-end/backend
   - Split driver: Simple front-end in guest, full back-end in host
2. Emulation
   - Emulate complete device behavior in software
3. Hardware assisted
   - Hardware directly supports device sharing (SR-IOV, VT-d)

Approach 1: Front-end/back-end driver model

Idea: Split drivers between guest and trusted domain

- Guest uses para-virtualized front-end to send reqs to backend driver
- Backend driver does actual IO and notifies to guest on completion

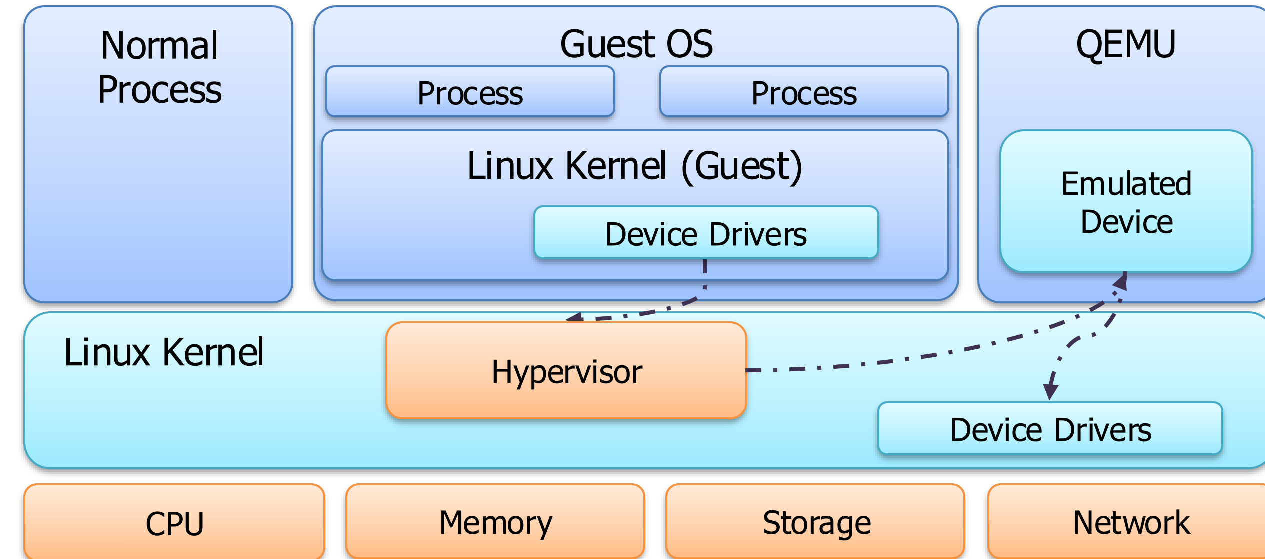
- Example: virtio in KVM, blkfront/blkback in Xen



EPFL

Approach 2: Device emulation

- Behavior emulated in software
- Guest uses native device driver
- VMM intercepts all access by configuring MMIO/PIO regions
- VMM forwards access to the emulated device



- Emulator performs real IO through host drivers and injects virtual interrupts

Example: Qemu

EPFL

Approach 3: Hardware-assisted IO virtualization

Idea: Hardware directly supports device sharing among VMs

- Guest uses unmodified device driver
- SR-IOV (Single root IO virtualization)
  - Physical devices exposes virtual functions
  - Each VM gets direct access to VFs
  - Bypasses hypervisor for data path
- Intel VT-d / AMD-Vi (IOMMU):
  - Translates device DMA addresses (like EPT for devices)
  - Prevents VM from using DMA to access other VM's memory
  - Enables safe device pass through