香港中文大學
The Chinese University of Hong Kong

CSCI5550 Advanced File and Storage Systems

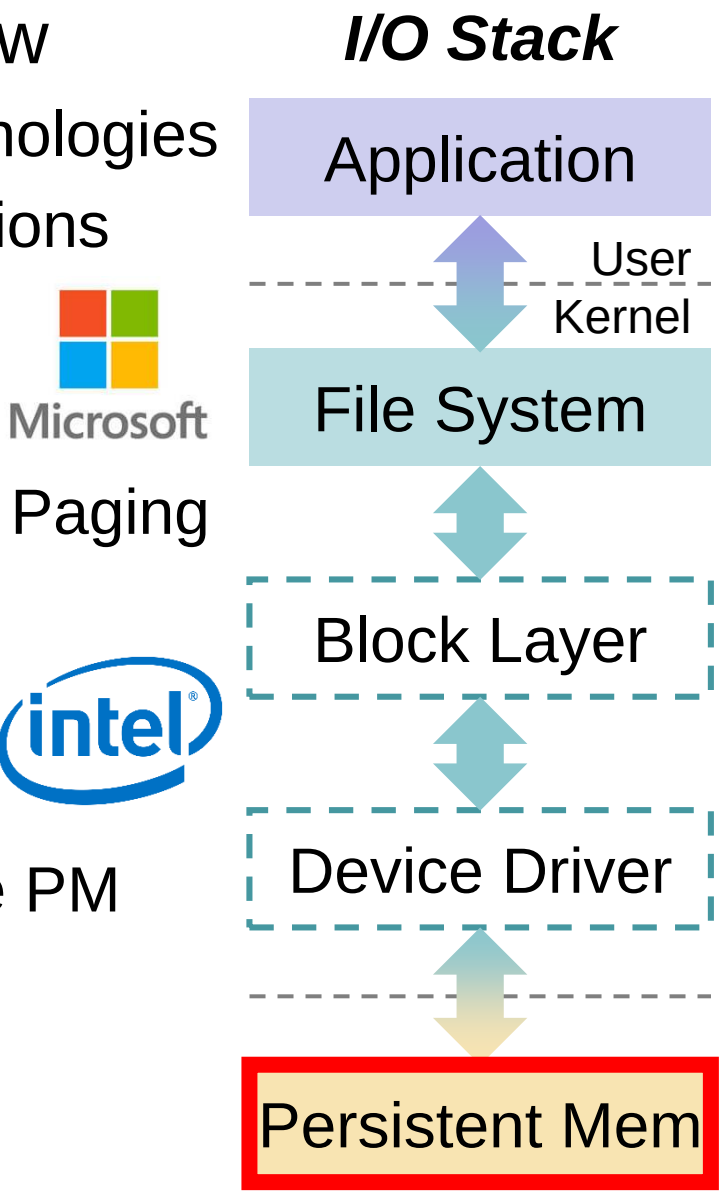
Lecture 08: Persistent Memory

Ming-Chang YANG

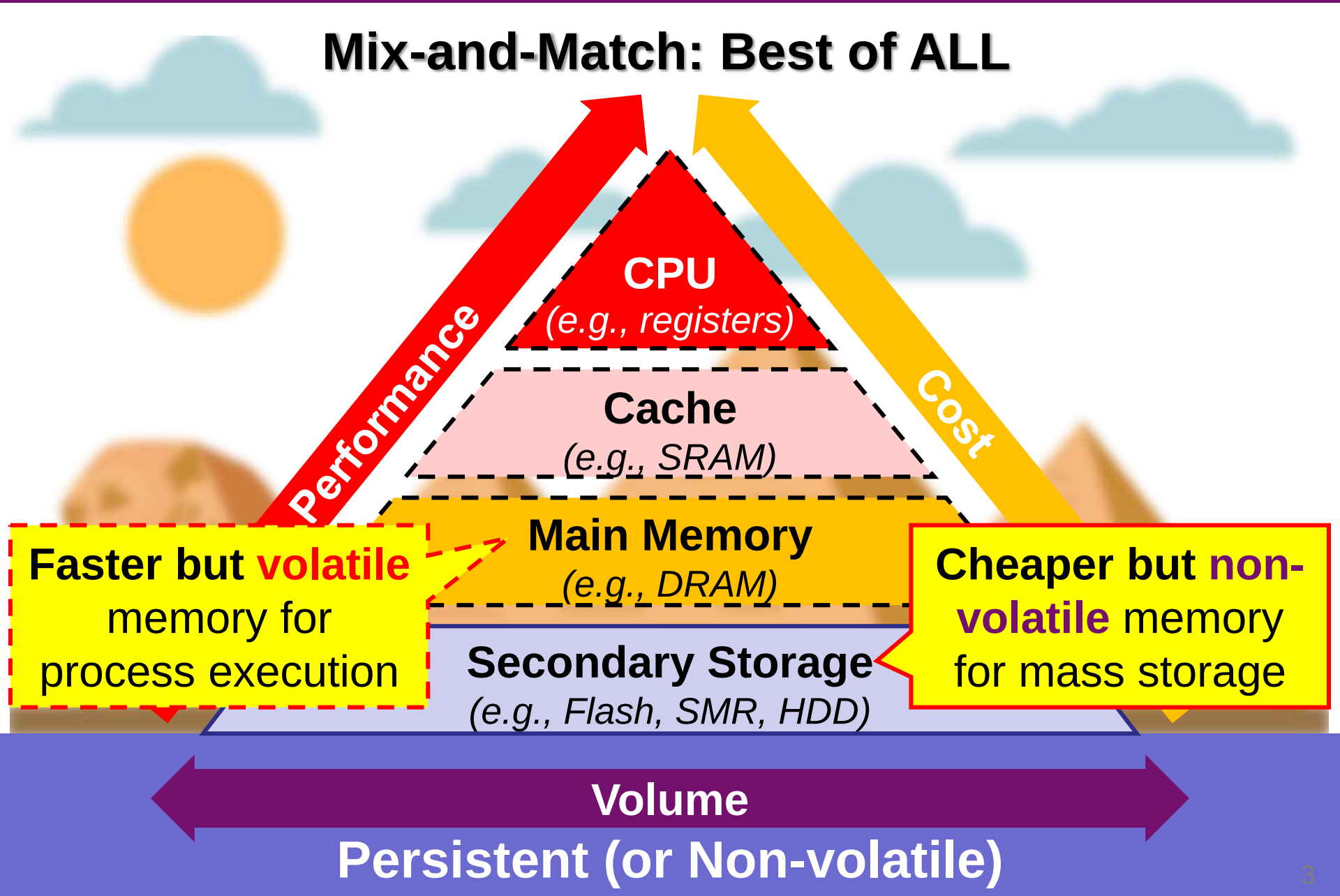
mcyang@cse.cuhk.edu.hk

Outline

- Persistent Memory: Why and How
  - Emerging Persistent Memory Technologies
  - Characteristics and Integration Options
- Byte-addressable Persistent FS
  - System Architecture
  - Consistency: Short-Circuit Shadow Paging
  - Write Ordering: Epoch Barriers
- Persistent Memory FS
  - System Architecture
  - Optimizations for Byte-Addressable PM
  - Hybrid Approach for Consistency
  - Protection from Stray Writes
  - Write Ordering and Durability



Preface: Memory Hierarchy Pyramid

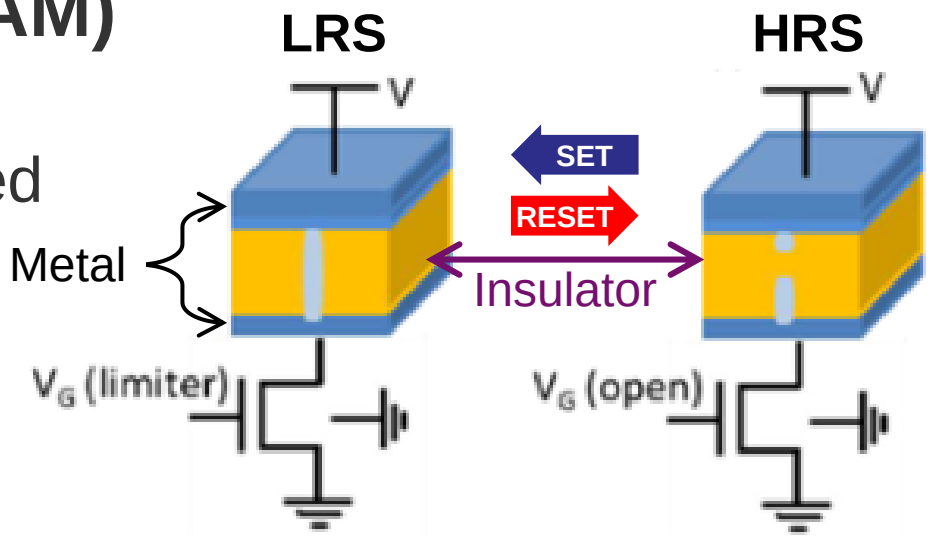


Why Persistent Memory

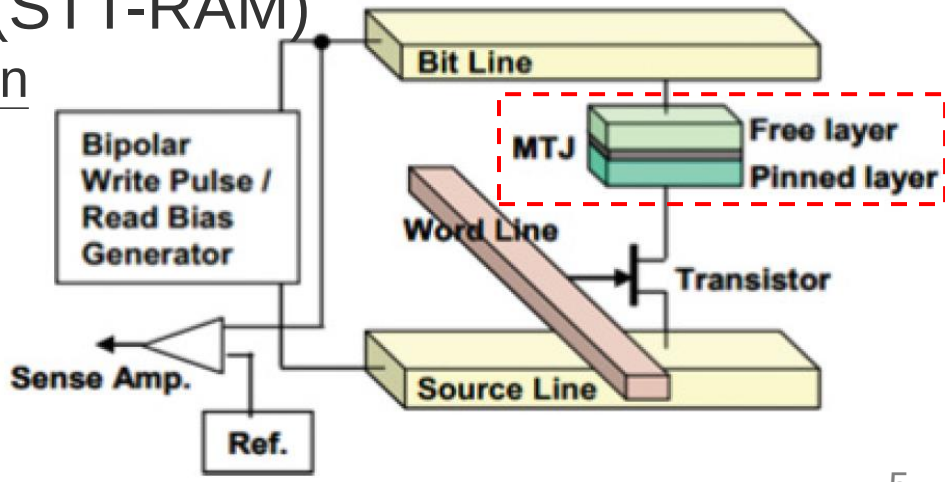
- **Persistent memory (PM)** might revolutionize the memory hierarchy due to "double-faced" advantages:
  - **Byte-addressability** as main memory;
  - **Comparable performance** as main memory;
  - Much **higher density** than main memory;
  - **Almost zero static power** consumption;
  - **Non-volatility** as secondary storage.
- PM has several implications on both OS and FS.
  - A large amount of work must be carried out.
- PM is just a collective term.
  - It is often called **non-volatile memory (NVM)** as well.
  - Many different "memory technologies" have emerged.
    - Flash memory is considered as a pioneer NVM.

Emerging Persistent Memories (1/2)

- **Resistive Memory (ReRAM)**
  - By changing the resistance value of the insulators based on the magnitude and polarity of applied voltage:
    - *Low Resistive State*
    - *High Resistive State*



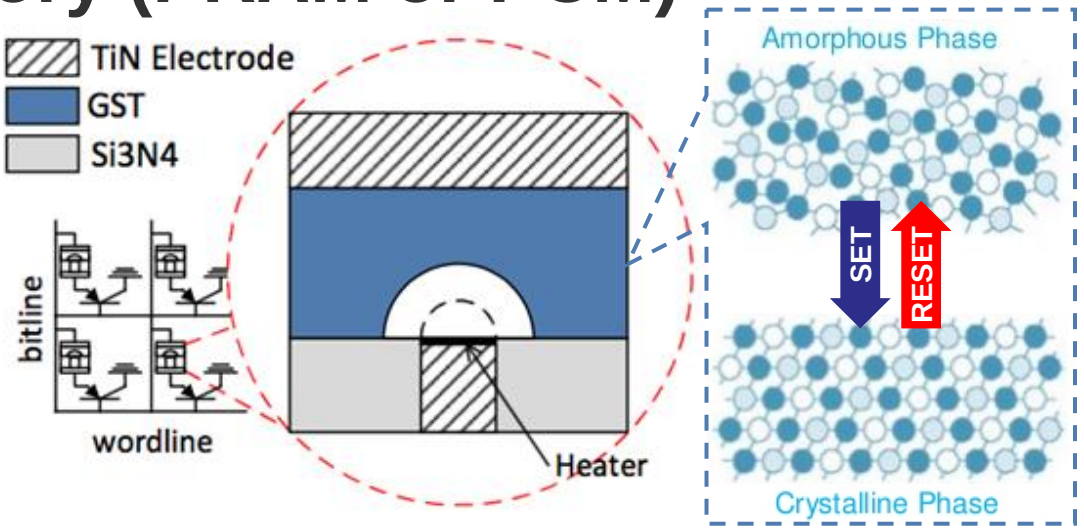
- **Magnetoresistive RAM (MRAM)**
  - Spin Torque Transfer RAM (STT-RAM)
    - By changing the magnetization direction of the free layer with high positive or high negative voltage difference:
      - *Same as the pinned layer*
      - *Opposite as the pinned layer*



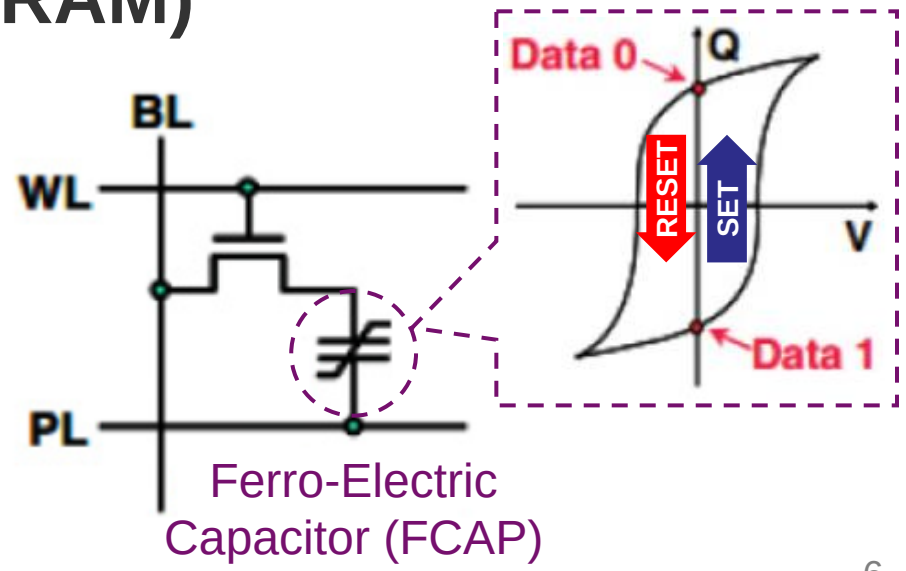
Emerging NVM: A Survey on Architectural Integration and Research Challenges (TODAES'17)

Emerging Persistent Memories (2/2)

- **Phase Change Memory (PRAM or PCM)**
  - By heating the phase change material with different *temperatures and time durations*:
    - *Amorphous Phase*
    - *Crystalline Phase*



- **Ferroelectric Memory (FeRAM)**
  - By characterizing the ferro-electric capacitor (FCAP) as two remnant reversible polarization states:
    - *Positive Remnant Polarization*
    - *Negative Remnant Polarization*



Emerging NVM: A Survey on Architectural Integration and Research Challenges (TODAES'17)

Characteristics of Memory Technologies

← Byte-Addressable → ← Block-Based →

	SRAM	STT-RAM	DRAM	Re-RAM	PCM	Fe-RAM	NAND Flash	HDD
Volatility	*V*	*NV*	*V*	*NV*	*NV*	*NV*	*NV*	*NV*
Cell Size (F^2)	$120 - 200$	$6 - 50$	$60 - 100$	$4 - 10$	$4 - 12$	$6 - 40$	$4 - 6$	N/A
Write Endurance	10^{16}	$10^{12} - 10^{15}$	$> 10^{15}$	$10^{8} - 10^{11}$	$10^{8} - 10^{9}$	$10^{14} - 10^{15}$	$10^{4} - 10^{5}$	$> 10^{15}$ ($* mech$)
Read Latency	$\sim 0.2 - 2\ ns$	$2 - 35\ ns$	$\sim 10\ ns$	$\sim 10\ ns$	$20 - 60\ ns$	$20 - 80\ ns$	$15 - 35\ us$	$3 - 5\ ms$
Write Latency	$\sim 0.2 - 2\ ns$	$3 - 50\ ns$	$\sim 10\ ns$	$\sim 50\ ns$	$20 - 150\ ns$	$50 - 75\ ns$	$200 - 500\ us$	$3 - 5\ ms$
Read Energy	*Low*	*Low*	*Medium*	*Low*	*Medium*	*Low*	*Low*	($* mechanical\ parts$)
Write Energy	*Low*	*High*	*Medium*	*High*	*High*	*High*	*Low*	
Static Power	*High*	*Low*	*Medium*	*Low*	*Low*	*Low*	*Low*	

Emerging NVM: A Survey on Architectural Integration and Research Challenges (TODAES'17)

Double-Faced Characteristics of PMs

- **Strengths**
  - ✓ Byte-addressability (*as SRAM and DRAM*)
    - *Each byte can be individually addressed.*
  - ✓ Bit-alterability (*as SRAM and DRAM*)
    - *Each cell can be individually altered.*
  - ✓ Non-volatility (*as SSD and HDD*)
  - ✓ Higher density (*than SRAM and DRAM*)
  - ✓ Almost zero static power
- **Shortcomings**
  - ✓ Less **write** endurance (than SRAM and DRAM)
  - ✓ Asymmetric read-**write** latency
  - ✓ Asymmetric read-**write** energy consumption

Write is the common Achilles heels of PMs!

PM Integration Options

- PM can be integrated into the memory hierarchy:
  - **Horizontally**: Supplement a given memory in the hierarchy.
  - **Vertically**: Shift down a given memory, or even replace it.
- PM can be exploited as:
  - **Processor Cache**
    - **L1 or L2 cache** is accessed at a high frequency: Most PMs are hard to offer very low latency and very high endurance is required.
    - **Last-level caches** are designed to reduce off-chip data movements: Most PMs can offer high density (to achieve large capacity of LLCs).
  - **Main Memory**
    - Most PMs can offer very appealing performance for main memory.
  - **Secondary Storage**
    - Similar to the pioneer PM (i.e., flash memory), some PMs can also offer high density, low static power, and faster I/O performance.

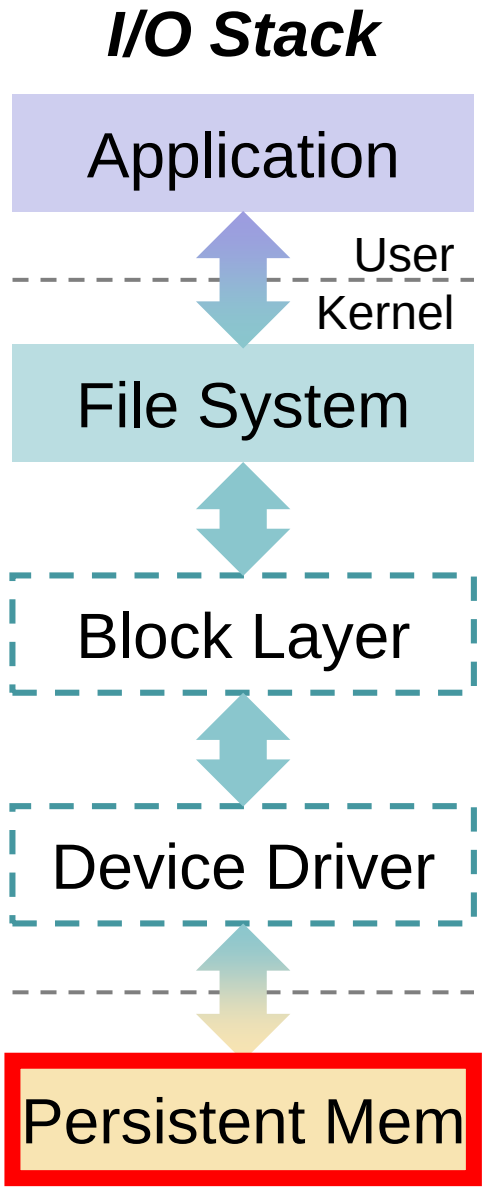
How to exploit PM to make storage systems better?

Outline

- Persistent Memory: Why and How
  - Emerging Persistent Memory Technologies
  - Characteristics and Integration Options
- Byte-addressable Persistent FS
  - System Architecture
  - Consistency: Short-Circuit Shadow Paging
  - Write Ordering: Epoch Barriers
- Persistent Memory FS
  - System Architecture
  - Optimizations for Byte-Addressable PM
  - Hybrid Approach for Consistency
  - Protection from Stray Writes
  - Write Ordering and Durability

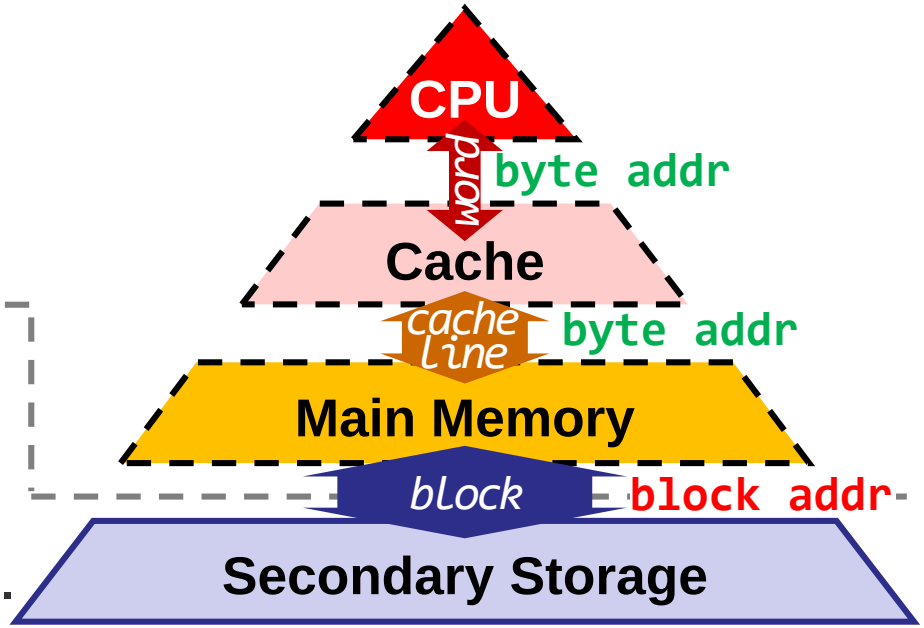
Microsoft

intel



Byte-addressable Persistent FS (BPFS)

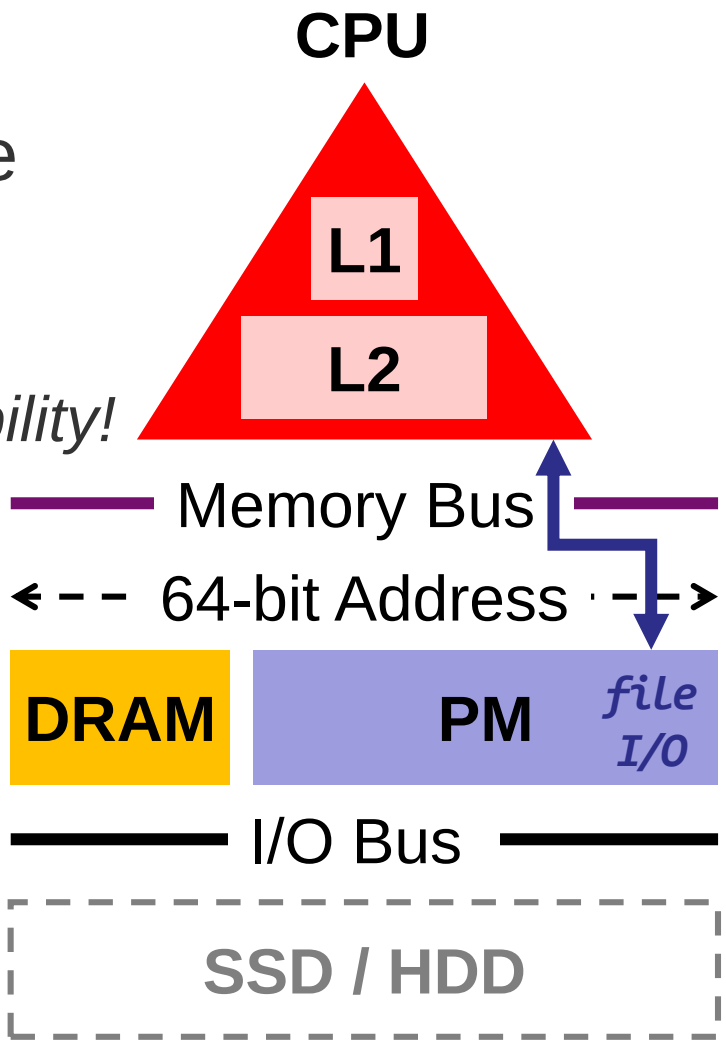
- Existing file/storage systems often face a trade-off among durability, consistency, and performance.
  - Data can be buffered in fast, byte-addressable but **volatile media** with the risk of losing data.
  - Data must be persisted on **non-volatile media**, but these devices support only slow and bulk data transfers.



- BPFS is designed to leverage the new persistent, **byte addressable** memory.
  - Fast, byte-addressable, and **non-volatile**!

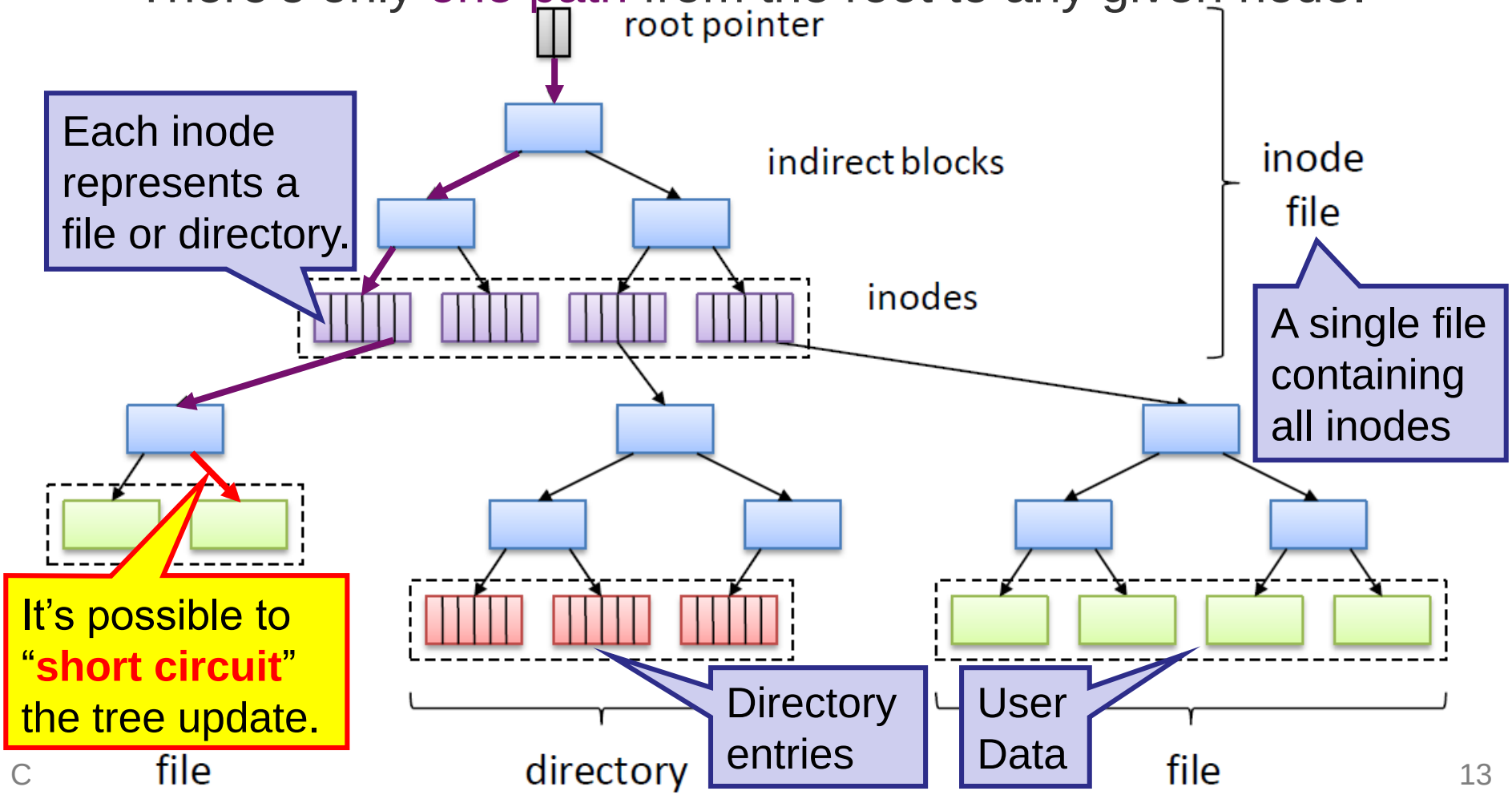
System Architecture of BPFS

- L1 and L2 caches are enabled normally for better performance.
- DRAM and PM are placed on the memory bus side-by-side.
  - ***Why?*** *Keeping PM behind I/O bus prohibits the use of byte-addressability!*
  - That is, both DRAM and PM are exposed directly to the CPU.
    - The 64-bit address is shared.
    - CPU can directly address PM with common load/store instructions.
- DRAM buffer cache is not used.
  - DRAM is used for other purposes.
- Persistent storage is not used.



File System Layout

- BPFS is composed **“trees”** of **fixed-size blocks**.
  - Fixed-sized blocks ease the allocation/deallocation of PM.
  - There’s only one path from the root to any given node.

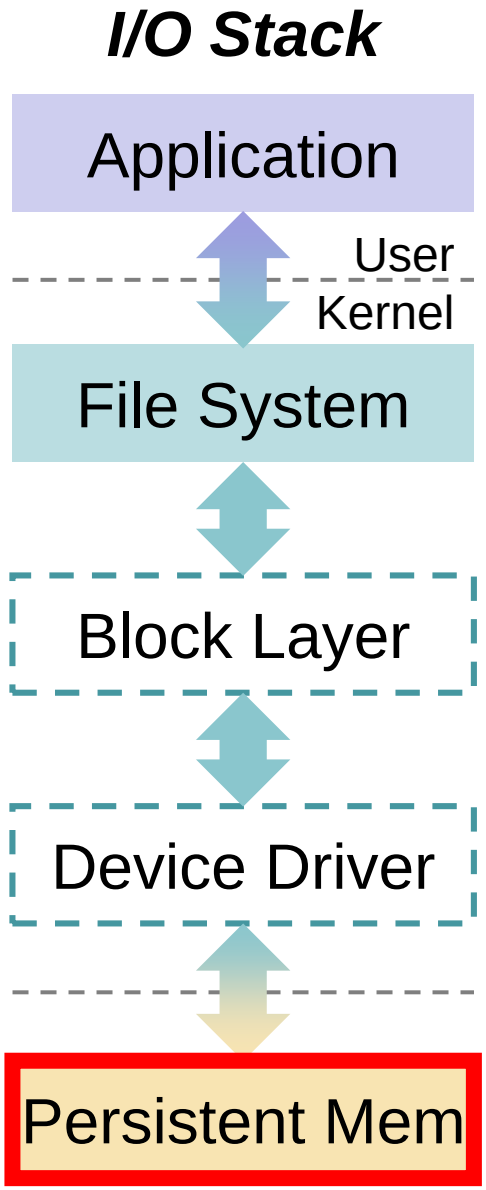


Outline

- Persistent Memory: Why and How
  - Emerging Persistent Memory Technologies
  - Characteristics and Integration Options
- Byte-addressable Persistent FS
  - System Architecture
  - Consistency: Short-Circuit Shadow Paging
  - Write Ordering: Epoch Barriers
- Persistent Memory FS
  - System Architecture
  - Optimizations for Byte-Addressable PM
  - Hybrid Approach for Consistency
  - Protection from Stray Writes
  - Write Ordering and Durability

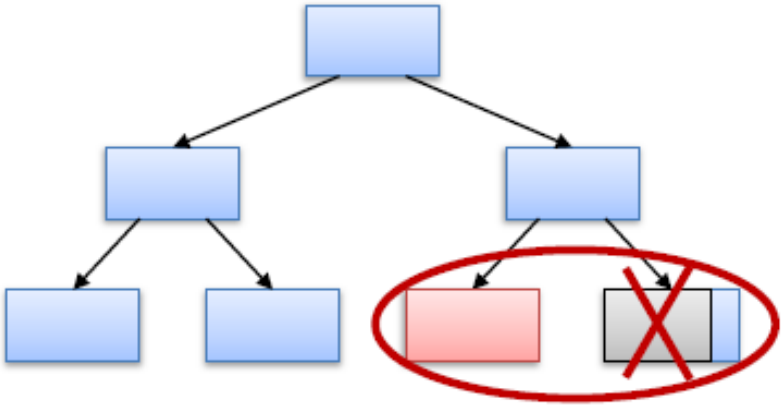
Microsoft

intel



Enforcing FS Consistency

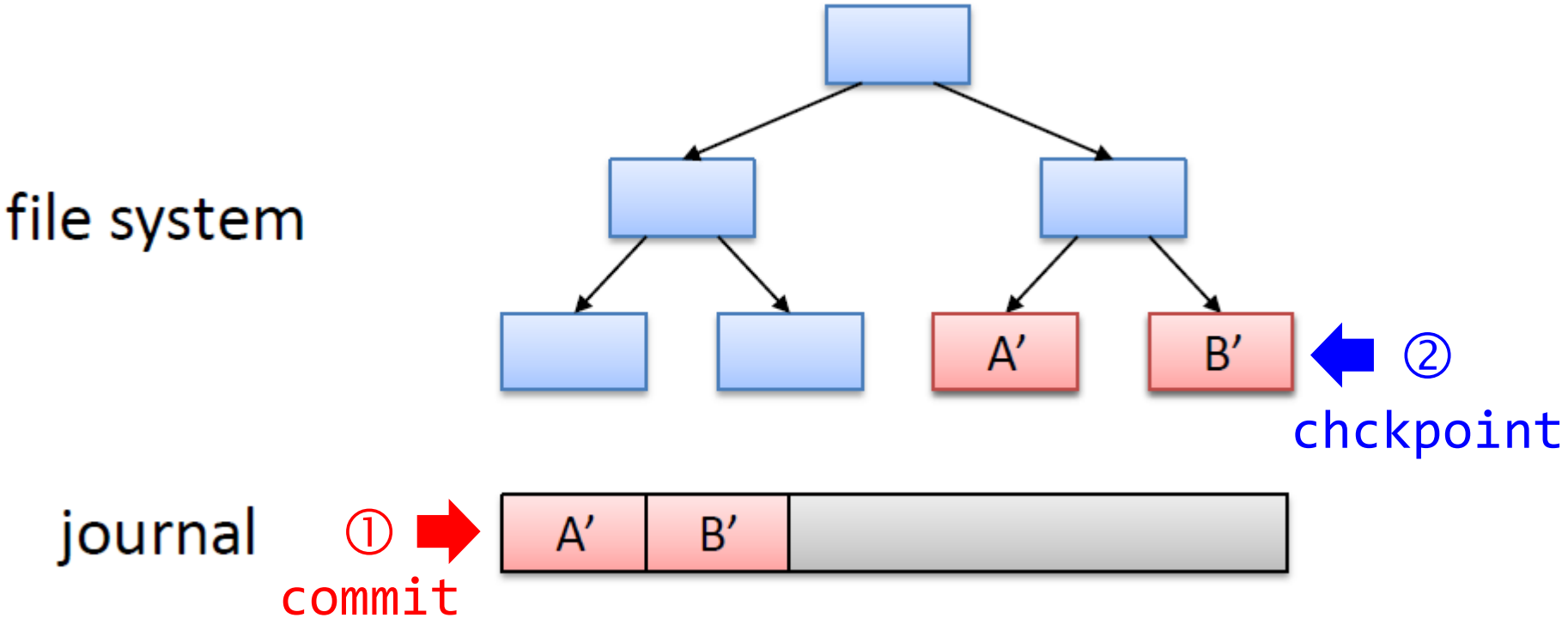
- What happens if FS crashes during an update?



- Most file/storage systems ensure the **consistency** of updates by two techniques:
  - ① **Write-ahead Logging (**or **Journaling)**
    - Write updates to a reserved and separate area (often as a sequential **log file**) before updating the corresponding locations.
  - ② **Shadow Paging**
    - Use **copy-on-write** (or **out-of-place update**) to perform updates in different locations, making the original data untouched.

Review #1: Journaling (1/2)

- **Idea**: ① Write to journal, then ② write to file system



- **Problem**: Consistent, but all data is **written twice**.
  - Most systems journal only metadata, but problem still exists!

Review #1: Journaling (2/2)

Data Journaling

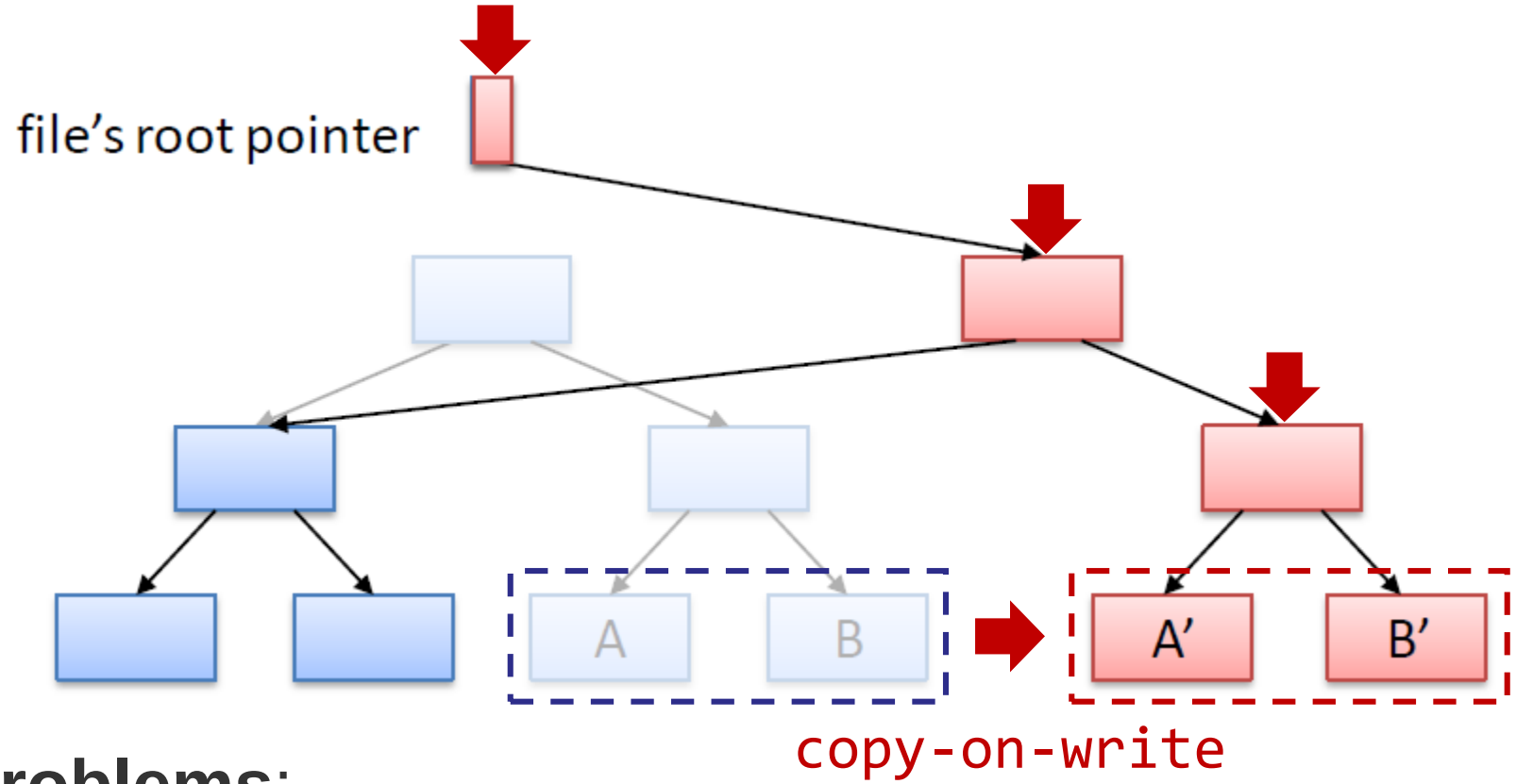
TxB	Metadata	Data	TxE	Metadata	Data
Issue	Issue	Issue			
Complete	Complete	Complete			
			Issue		
			Complete		
				Issue	Issue
				Complete	Complete

Metadata Journaling

TxB	Metadata	TxE	Metadata	Data
Issue	Issue			Issue
Complete	Complete			Complete
		Issue		
		Complete		
			Issue	
			Complete	

Review #2: Shadow Page

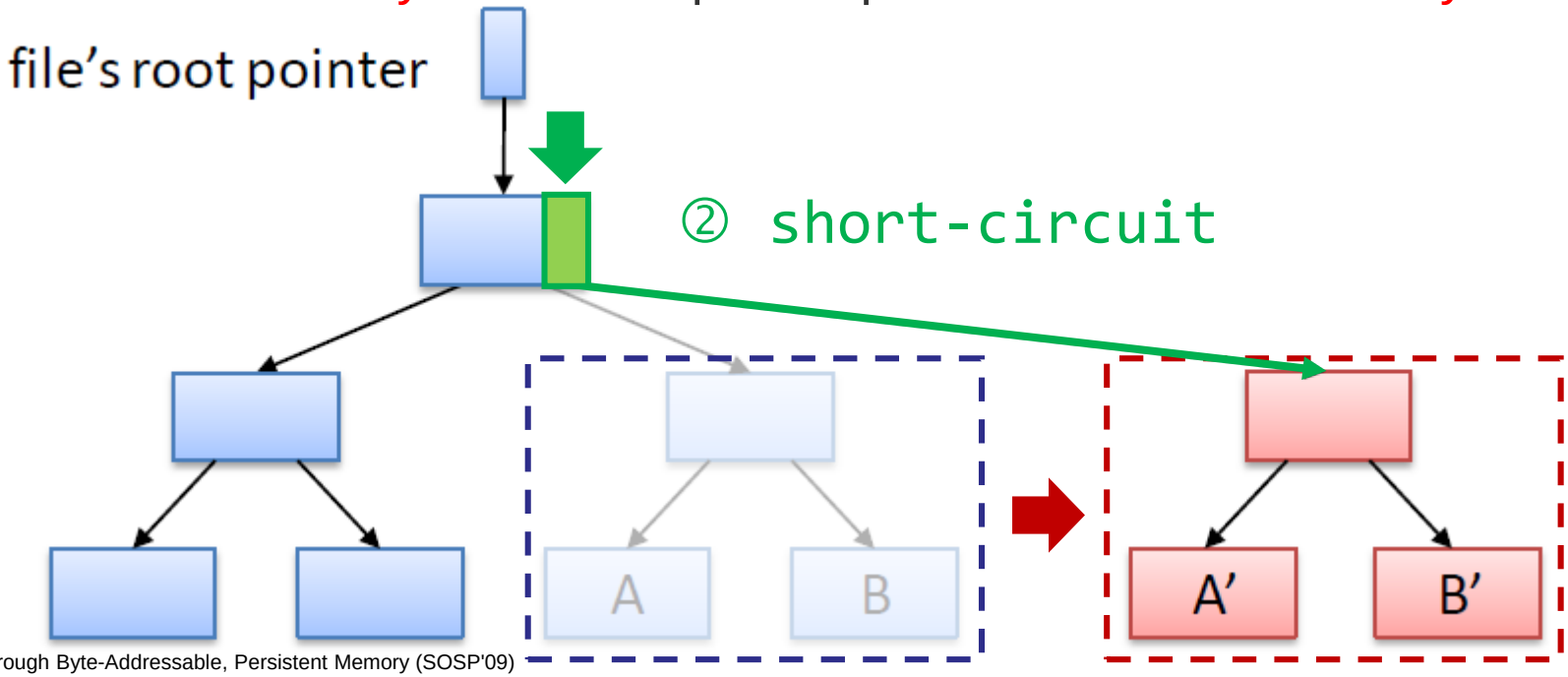
- **Idea**: Use copy-on-write up to root of the file system



- **Problems**:
  - Any change requires "**bubbling-up**" of updates to the root.
  - Even small writes require **large copying overhead**.

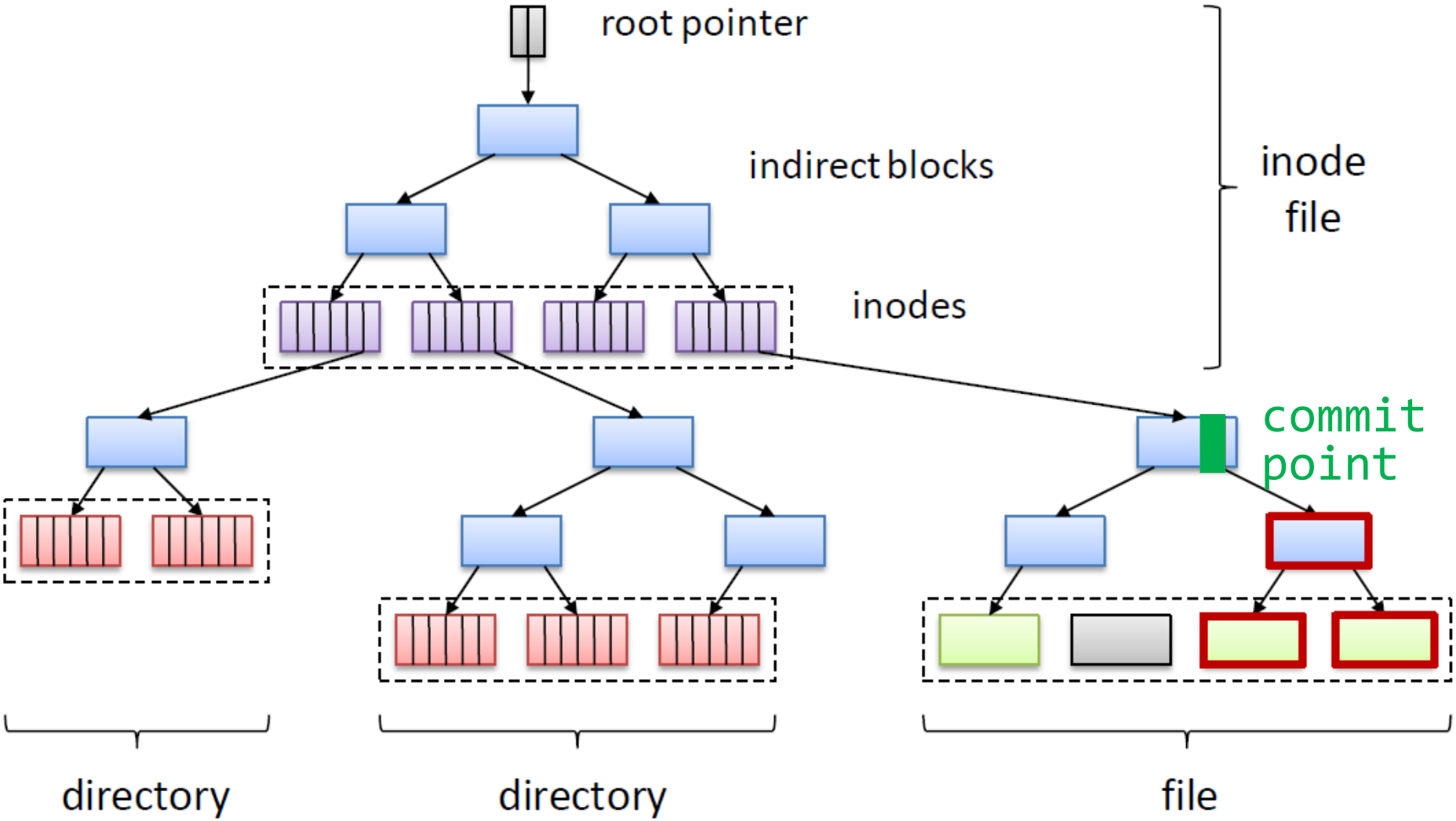
BPFS: Short-Circuit Shadow Paging (1/3)

- BPFS leverages the **byte-addressability** of PM to implement the "short-circuit" shadow paging.
  - **Idea**: Perform atomic in-place update for **small writes** to avoid (or "short-circuit") propagating copies to the root.
    - Costly copy-on-writes can be restricted to a small subtree of FS.
    - The atomicity of small in-place update must be ensured by hardware.



Better I/O Through Byte-Addressable, Persistent Memory (SOSP'09)

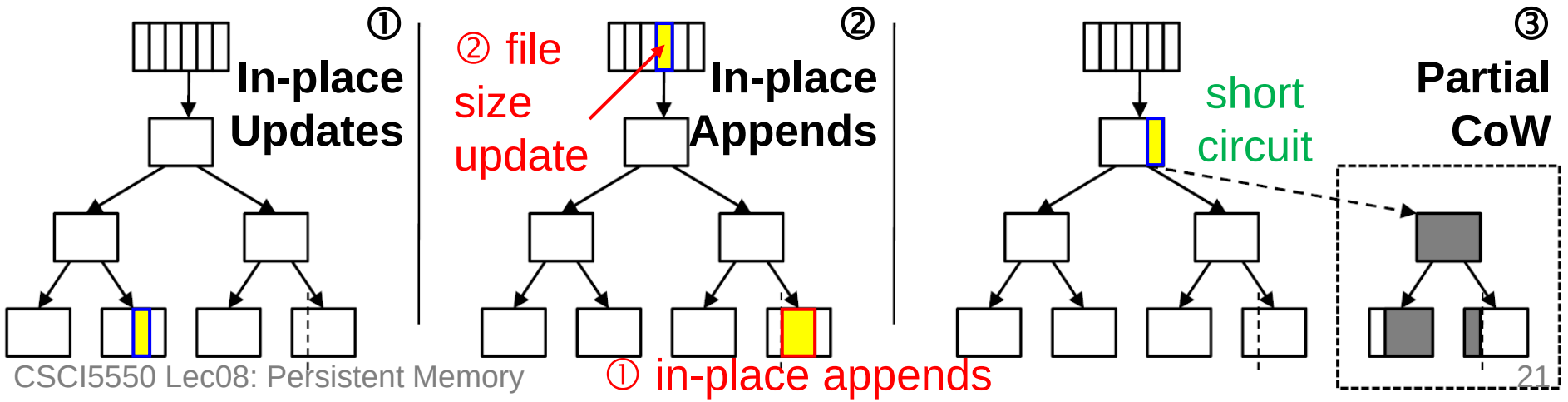
BPFS: Short-Circuit Shadow Paging (2/3)



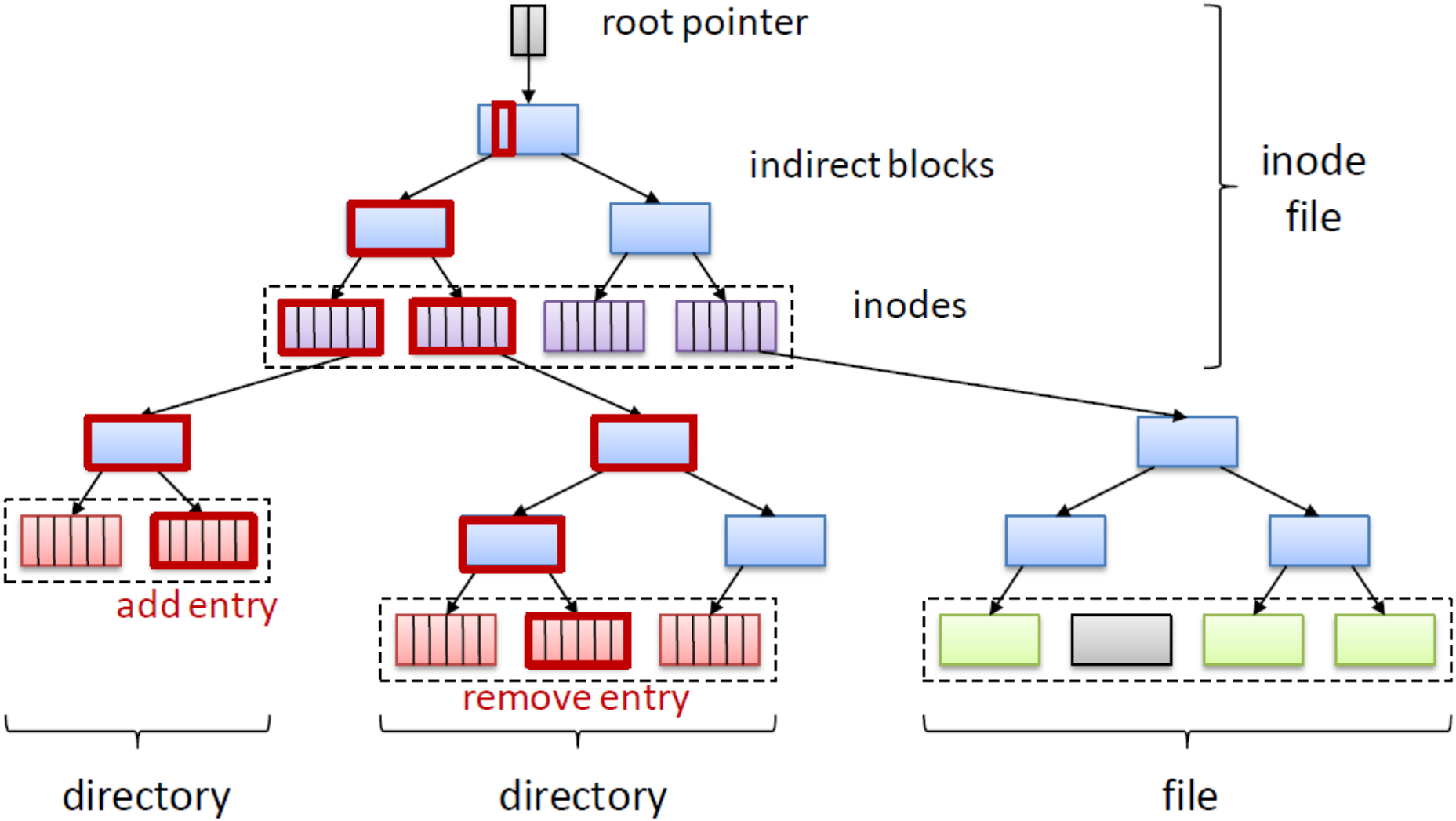
- BPFS does not need to copy the entire tree "above" the commit point (i.e., short-circuit point).

BPFS: Short-Circuit Shadow Paging (3/3)

- SCSP guarantees the consistency of data updates by three distinct methods via atomic in-place update:
  - ① **In-place Update**: Perform **64-bit (8-byte)** atomic in-place write at highest efficiency for both metadata and data.
    - How to ensure atomicity? Augmenting a capacitor on PM.
  - ② **In-place Append**: Perform in-place appends *regardless of the size*, then atomically "commit" the file size in the inode.
  - ③ **Partial Copy-on-Write**: Perform regular copy-on-writes until the write can be "short-circuited" by an atomic update.



SCSP Limitation



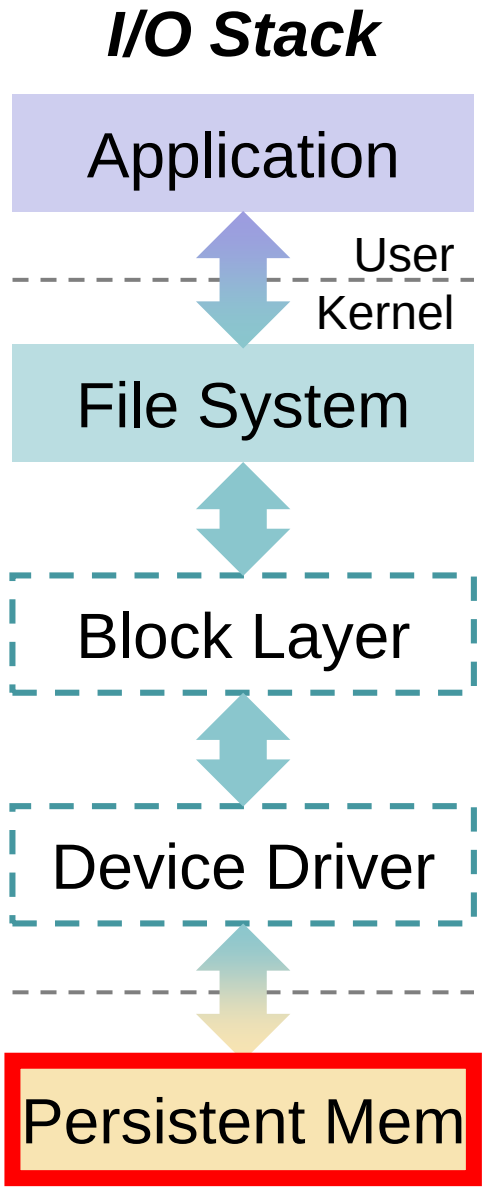
- **Cross-directory rename** may still bubble the copy-on-writes up to the common ancestor (even the root!).

Outline

- Persistent Memory: Why and How
  - Emerging Persistent Memory Technologies
  - Characteristics and Integration Options
- Byte-addressable Persistent FS
  - System Architecture
  - Consistency: Short-Circuit Shadow Paging
  - Write Ordering: Epoch Barriers
- Persistent Memory FS
  - System Architecture
  - Optimizations for Byte-Addressable PM
  - Hybrid Approach for Consistency
  - Protection from Stray Writes
  - Write Ordering and Durability

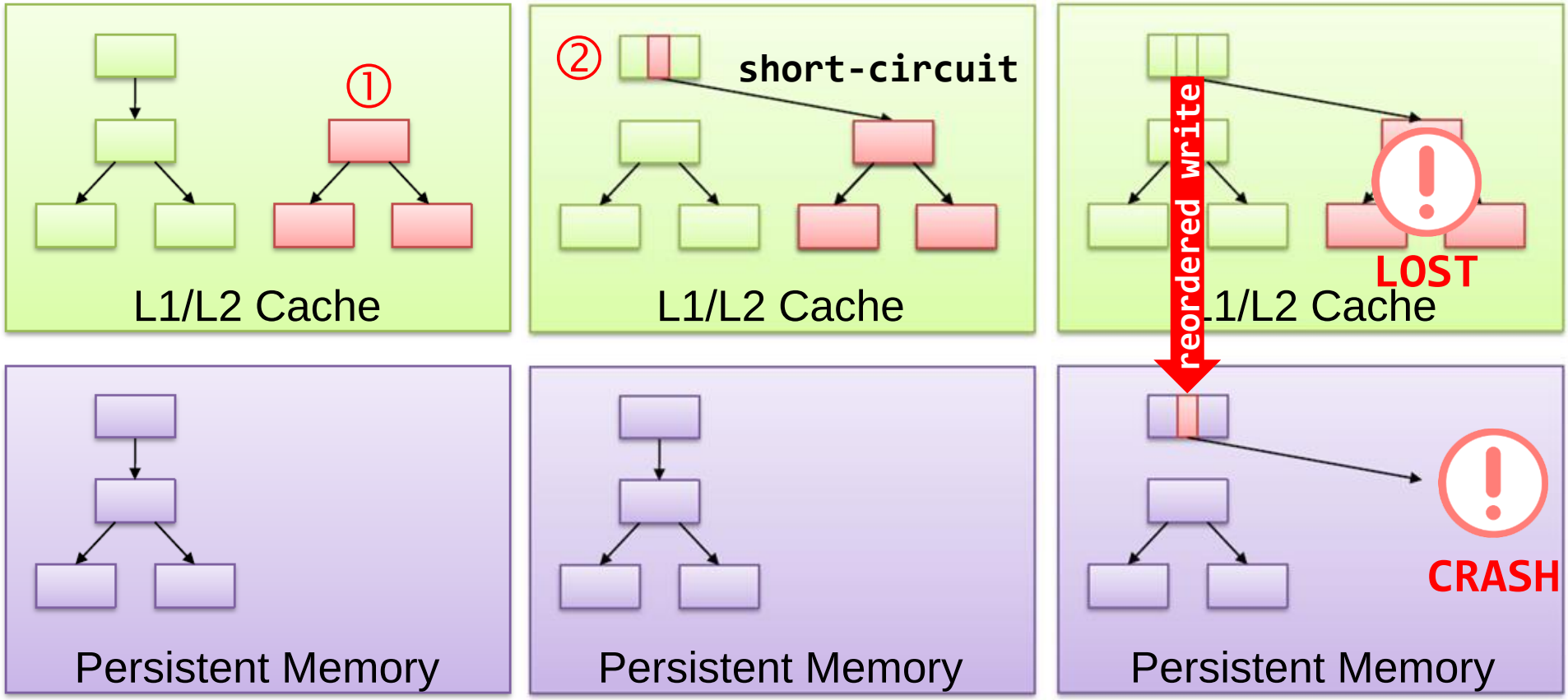
Microsoft

intel



Enforcing Ordering of Writes (1/2)

- Existing cache hierarchies and memory controllers may re-order writes for improve performance.
- However, the write order is crucial for PM consistency.



Enforcing Ordering of Writes (2/2)

- **Possible Solutions**: Costly in terms of performance!

① Uncached PM

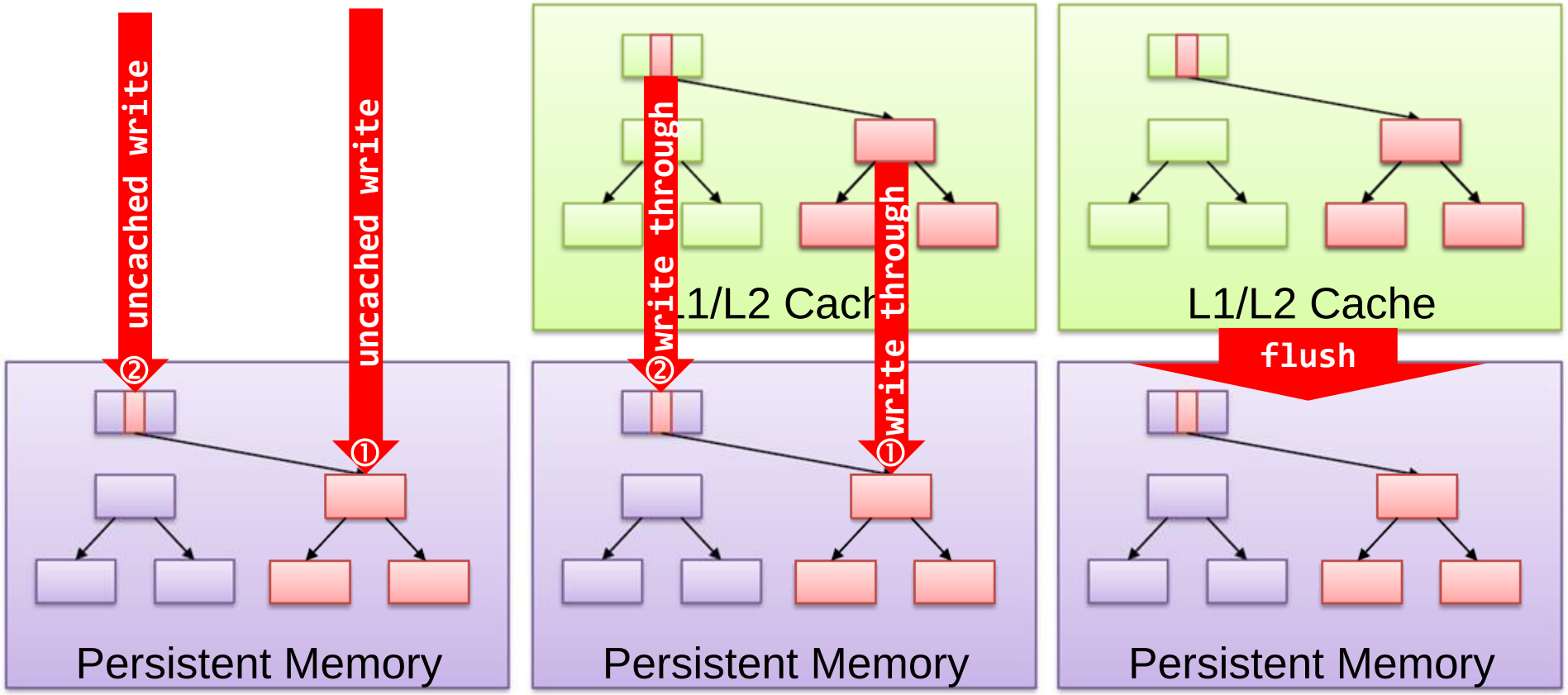
- Disable all cache(s)

② Write Through

- Update cache and memory concurrently

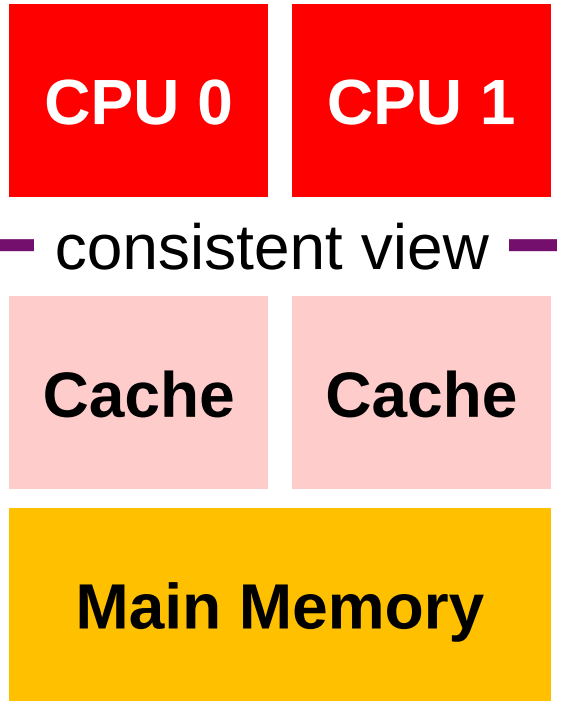
③ Cache Flush

- Flush entire cache at each memory barrier



Review: Memory Barrier

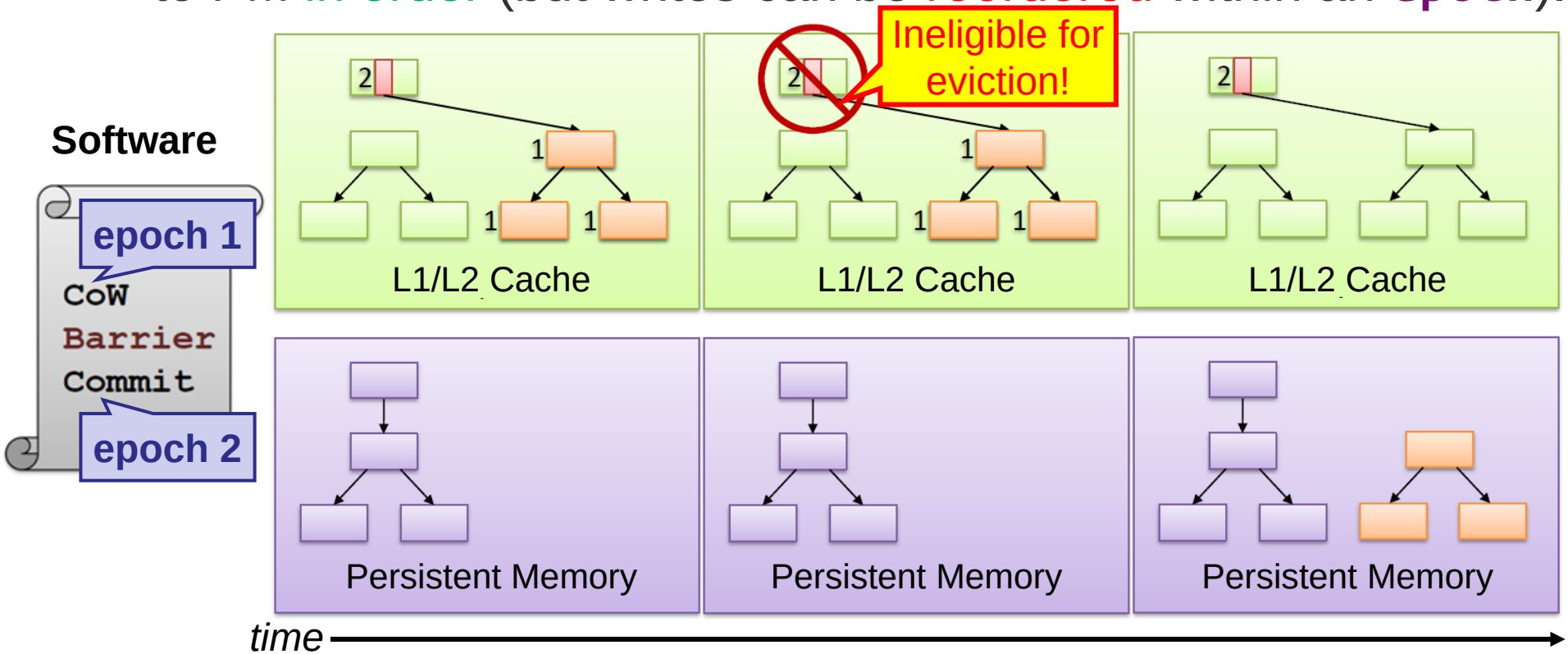
- **Memory barriers** (e.g., `x86 mfence`) ensure that all CPUs have a consistent view of memory.
  - It does not matter in what order data is actually written back to **memory**.
    - *After all, with the volatile memory, the ordering issue is irrelevant, while the cache coherence issue is critical.*



- Consider writes `A` and `B` are separated by an `mfence`:
  - The `mfence` only guarantees that `A` will be written to the **cache** (and made visible to all other CPUs via cache coherence) before `B` is written to the **cache**;
  - The `mfence` does not ensure that `A` will be written back to **memory** before `B` is written to the **memory**.

BPFS: Cache + Epoch Barriers (1/2)

- BPFS embraces division of labor between SW & HW:
  - **Software** issues **`write barriers`** that delimit a sequence of writes called an **`epoch`**;
  - **Hardware** guarantees that **`epochs`** are evicted from cache to PM in order (but writes can be reordered within an **`epoch`**).

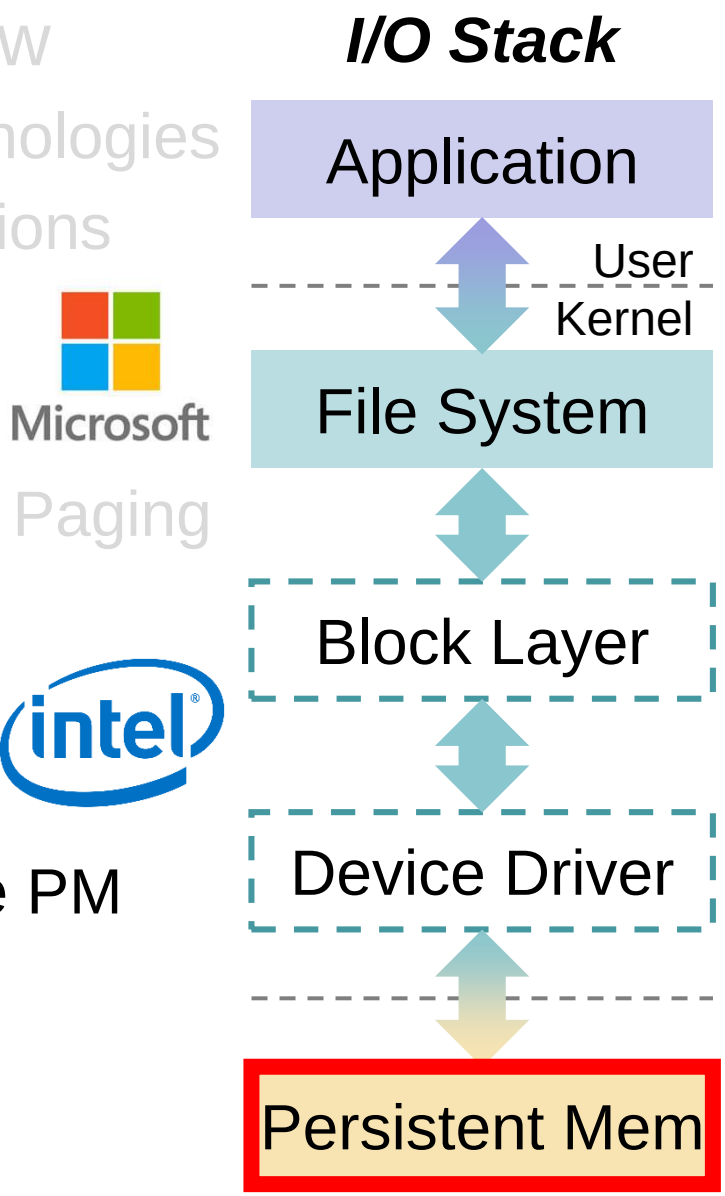


BPFS: Cache + Epoch Barriers (2/2)

- The hardware support for epoch must includes the following modifications to the PC architecture:
  ① Each **processor** must track the current epoch to maintain ordering among writes with an *epoch ID counter.*
    - It is incremented by one each time the processor commits an epoch.
  ② Each **cache line** is extended with the following two values:
    - A *reference bit* to indicate whether the cached data references PM.
    - An *epoch ID* to indicate the epoch to which this cache line belongs.
  ③ The **cache replacement logic** is also extended to ensure the evictions of cache lines are in epoch order strictly.
    - The cache controller tracks the oldest epoch in the cache, and considers cache lines of newer epochs to be ineligible for eviction.
  ④ The **memory controller** must ensure a write cannot be reflected to PM before all in-flight evictions are performed.

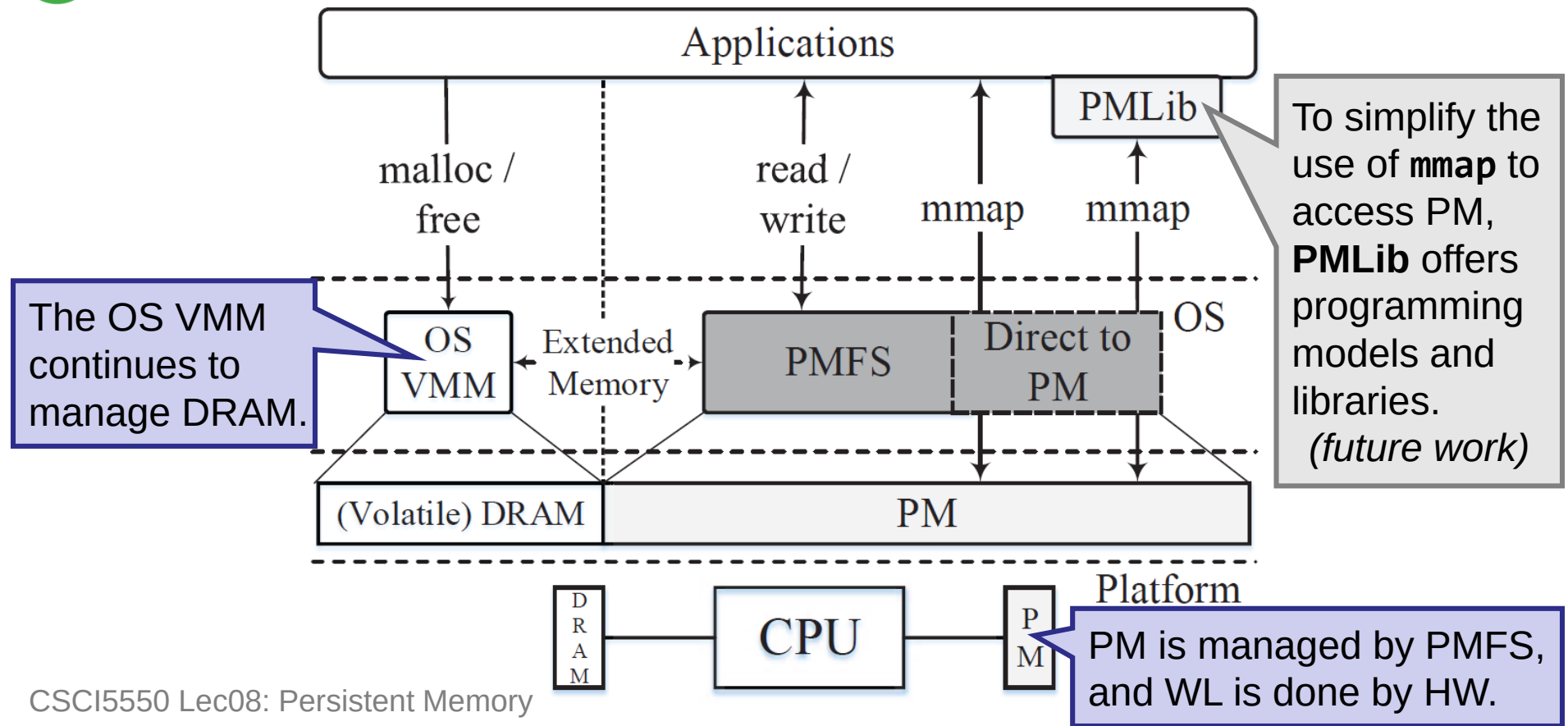
Outline

- Persistent Memory: Why and How
  - Emerging Persistent Memory Technologies
  - Characteristics and Integration Options
- Byte-addressable Persistent FS
  - System Architecture
  - Consistency: Short-Circuit Shadow Paging
  - Write Ordering: Epoch Barriers
- Persistent Memory FS
  - System Architecture
  - Optimizations for Byte-Addressable PM
  - Hybrid Approach for Consistency
  - Protection from Stray Writes
  - Write Ordering and Durability



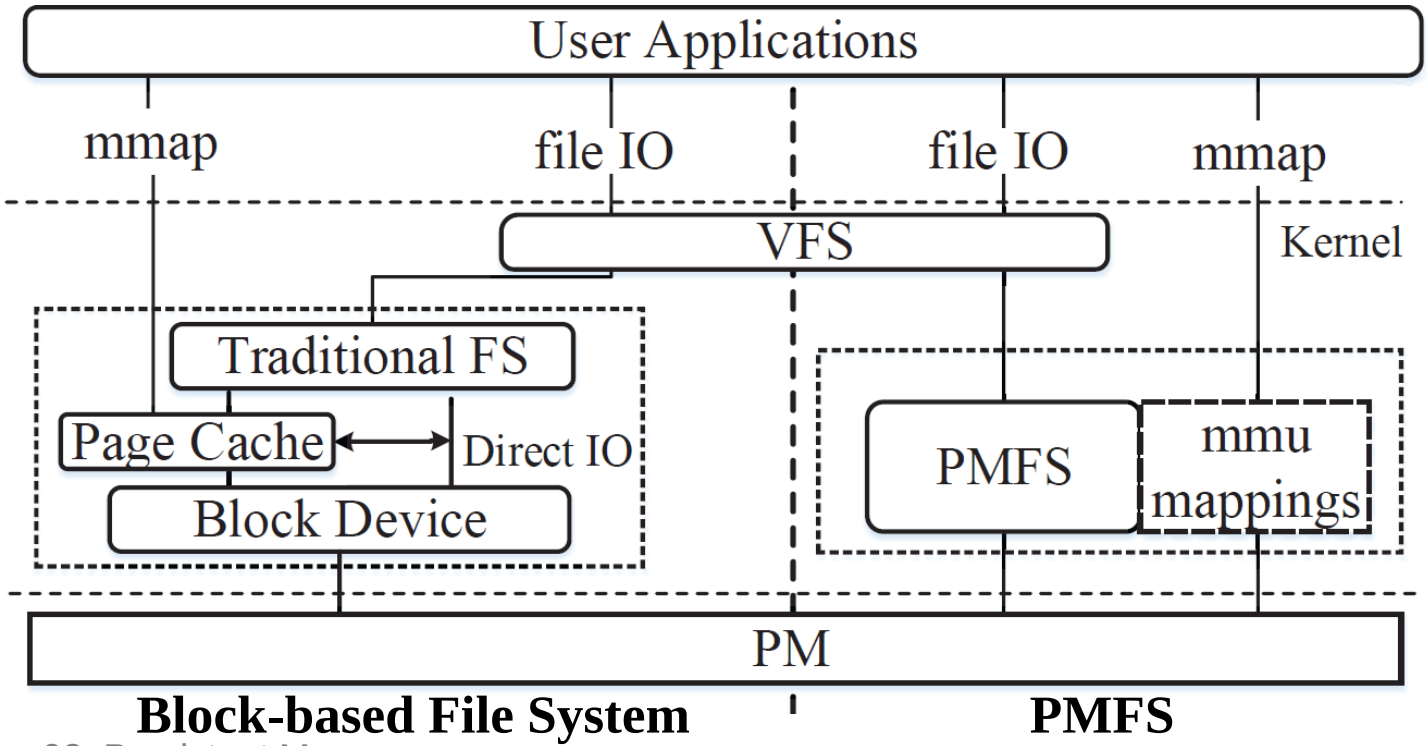
System Architecture of PMFS

- System software could manage PM in several ways:
  - ① Extending **virtual memory manger (VMM)** to manage PM
  - ② Utilizing PM as a **block device** with an existing file system
  - ③ Designing a **new file system** optimized for PM

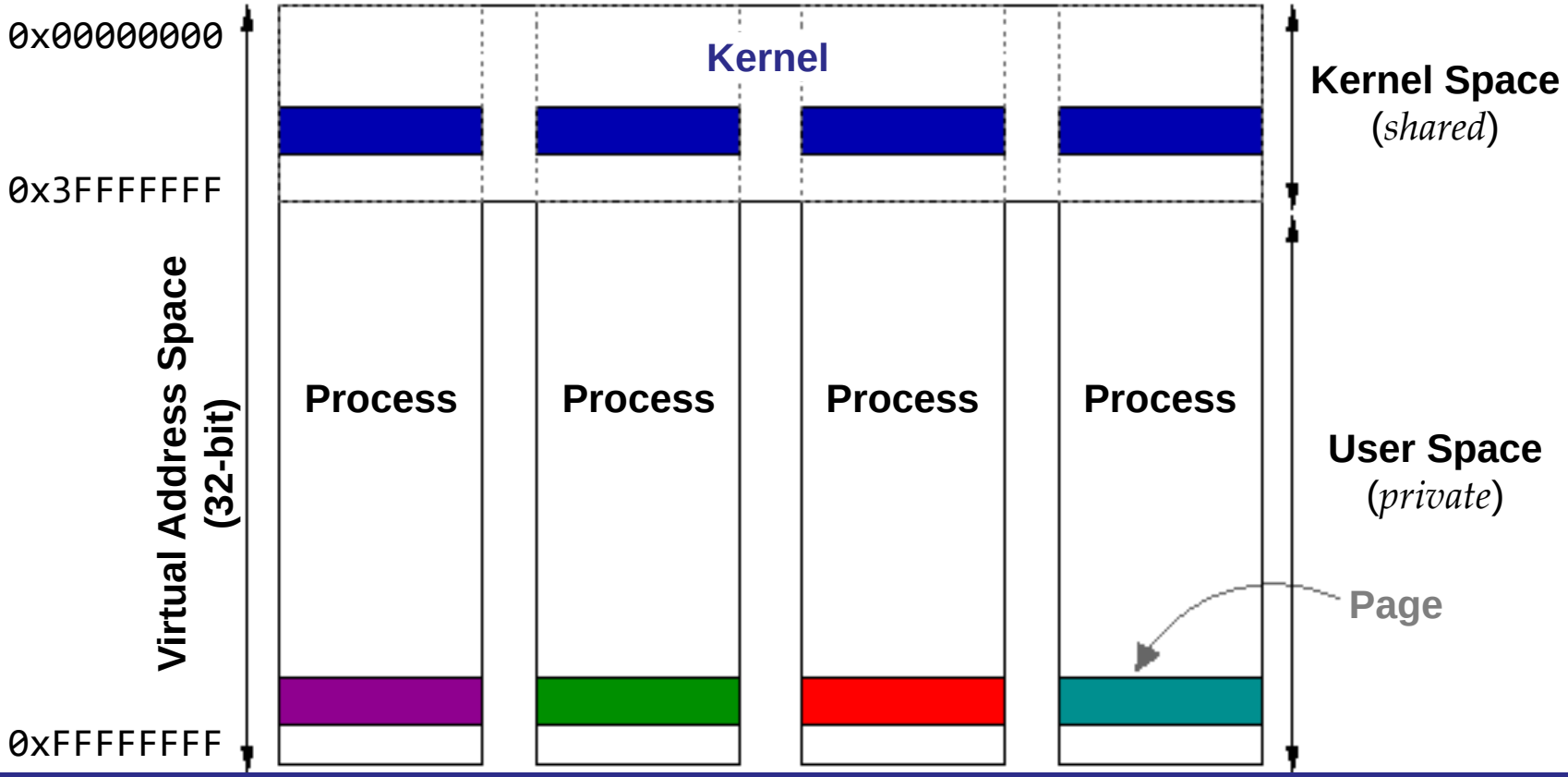


Traditional FS vs. PMFS

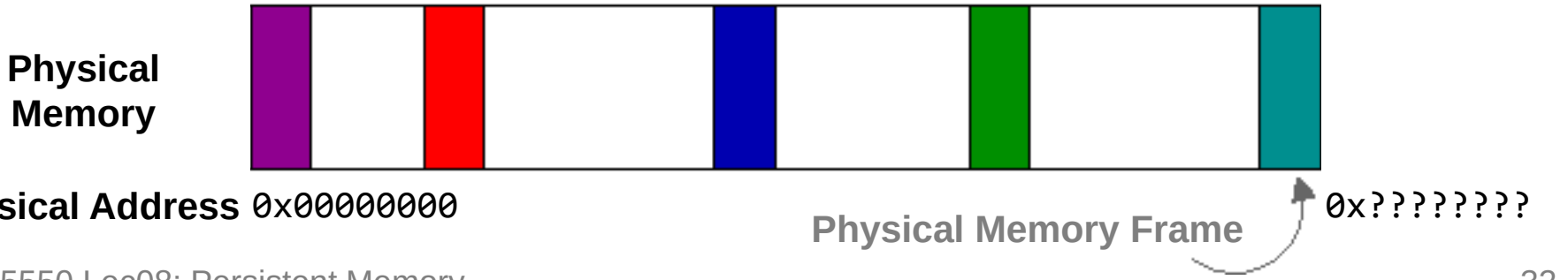
- PMFS eliminates copy overheads and offers essential benefits (up to $22X$) to legacy applications by:
  ① Exploiting the PM's byte-addressability;
  ② Avoiding the block layer; and
  ③ Mapping PM directly to address space of applications.



Review: Address Space in Linux

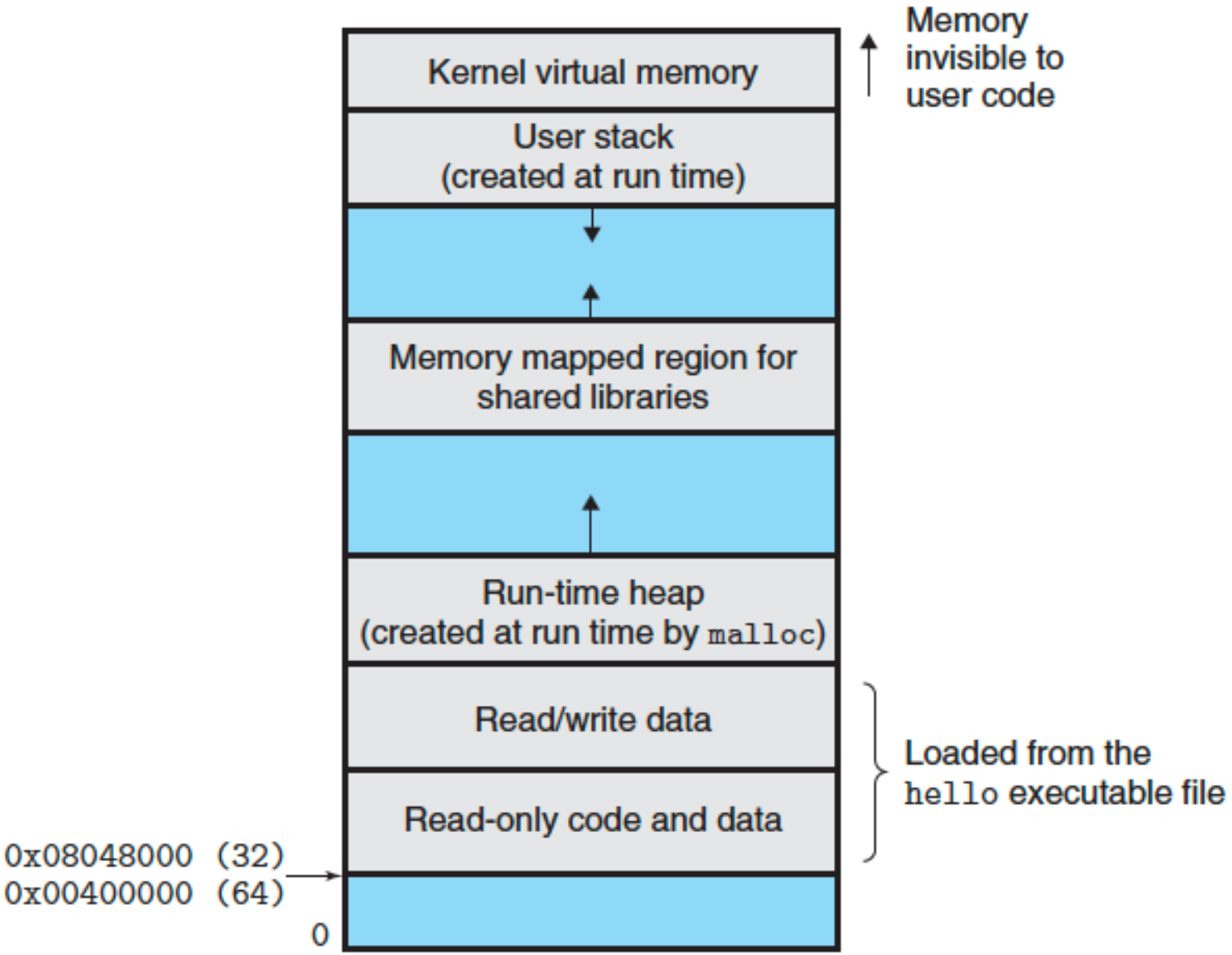


The **page table** keeps the mapping between virtual addresses and physical addresses.



https://blog.csdn.net/Takatsukii/article/details/41652127

Review: Process Address Space



https://stackoverflow.com/questions/9511982/virtual-address-space-in-the-context-of-programming?lq=1

Review: Transition between User/Kernel

① User code actively issues a system call.

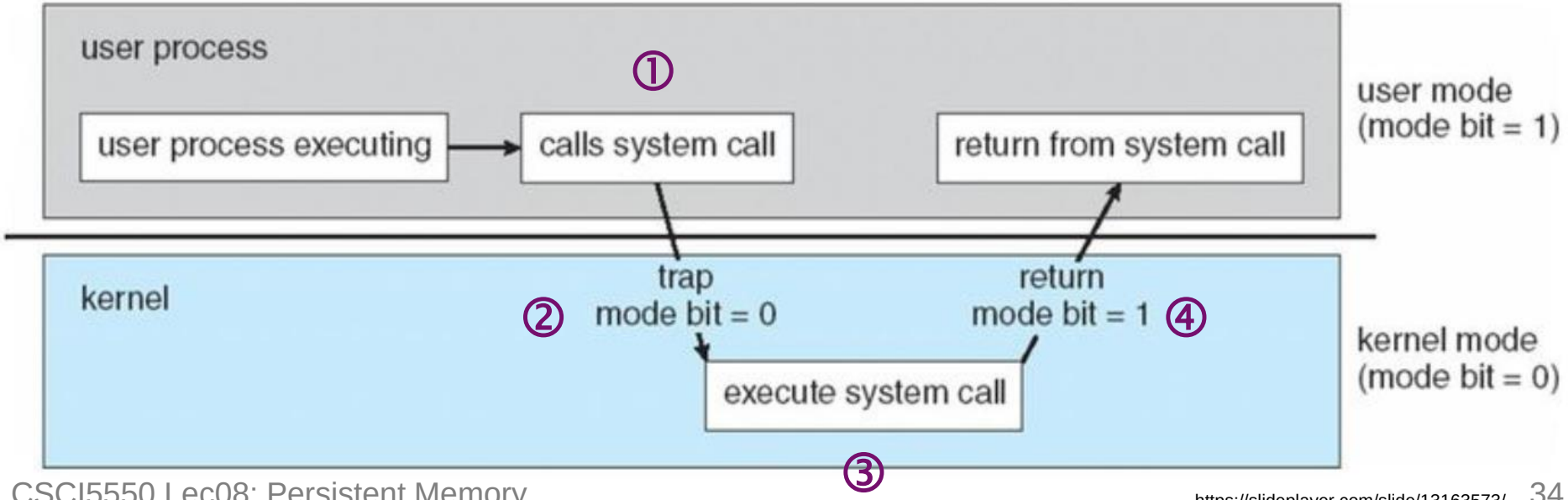
② The mode is changed to the kernel mode.

③ The system call handler (kernel code) is executed:

- ➢ Privileged instructions or shared kernel space may be used.

④ The mode is changed back to the user mode.

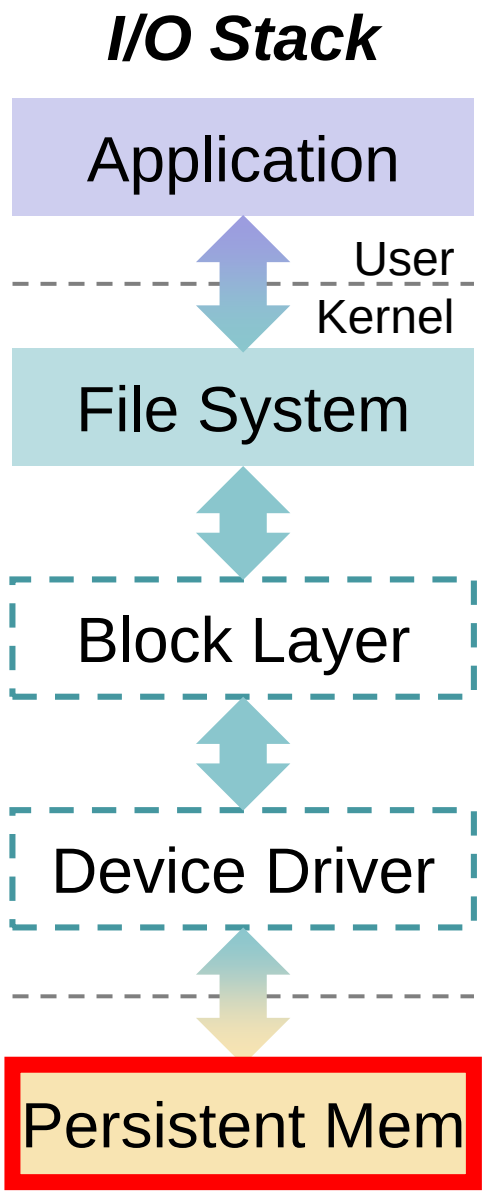
- *Interrupt/exceptions may also trigger mode transition.*



https://slideplayer.com/slide/13163573/

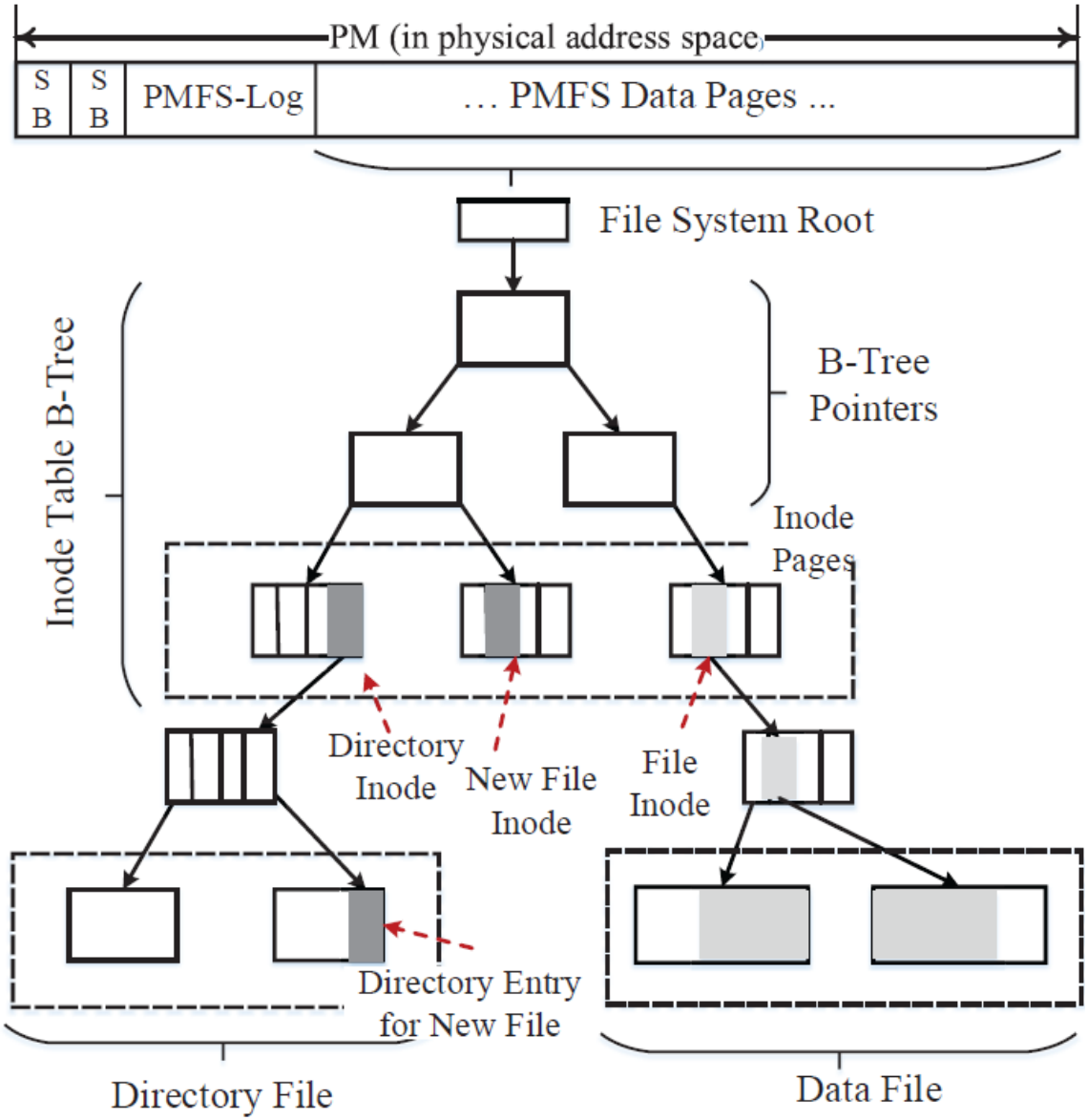
Outline

- Persistent Memory: Why and How
  - Emerging Persistent Memory Technologies
  - Characteristics and Integration Options
- Byte-addressable Persistent FS
  - System Architecture
  - Consistency: Short-Circuit Shadow Paging
  - Write Ordering: Epoch Barriers
- Persistent Memory FS
  - System Architecture
  - Optimizations for Byte-Addressable PM
  - Hybrid Approach for Consistency
  - Protection from Stray Writes
  - Write Ordering and Durability



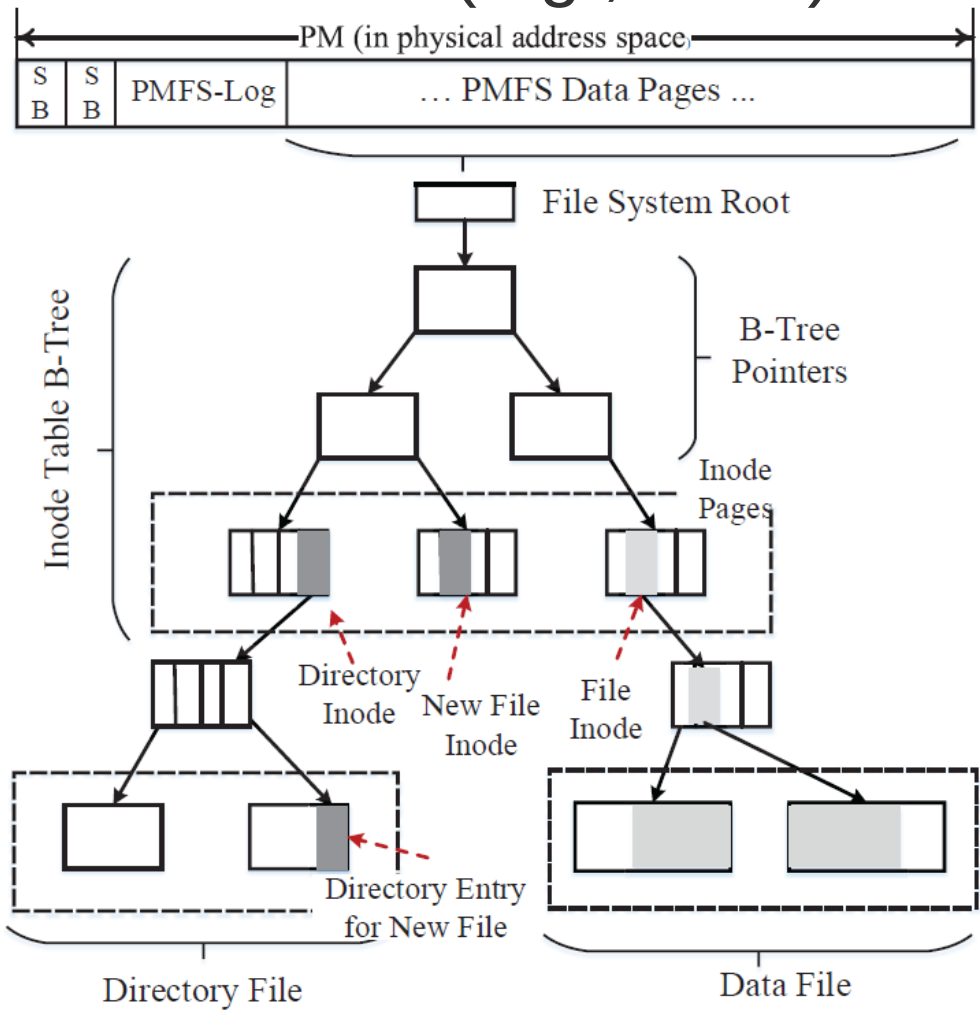
PM Optimizations: FS Layout on "PM"

- PM Address Space:
  - ① Superblock and its redundant copy
  - ② A journal area (called PMFS-Log)
  - ③ Dynamic allocated "pages"
- **B-trees** are used to organize the metadata:
  - ① Inode table
  - ② Directory inodes
  - ③ File inodes



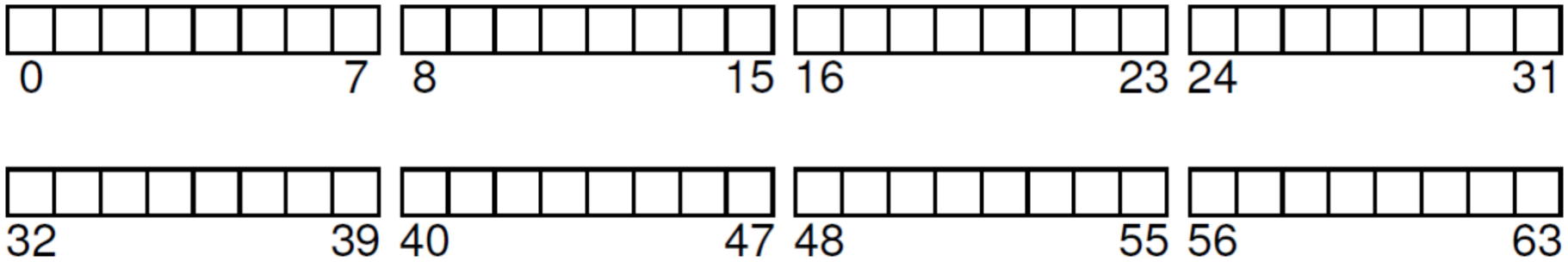
PM Optimizations: Allocation

- Modern FSs are *extent-based* (e.g., `ext4`, `btrfs`), while some older FSs are *indirect block-based* (e.g., `ext2`).
- Allocations in PMFS are page-based.
  - 4KB pages for **metadata** (i.e., internal nodes);
  - 4KB, 2MB, or 1GB pages for **data** (i.e., leaf nodes).
    - See paper for more detailed discussions.
- PMFS coalesces adjacent pages to avoid major fragmentation.



Recall: Block Allocation

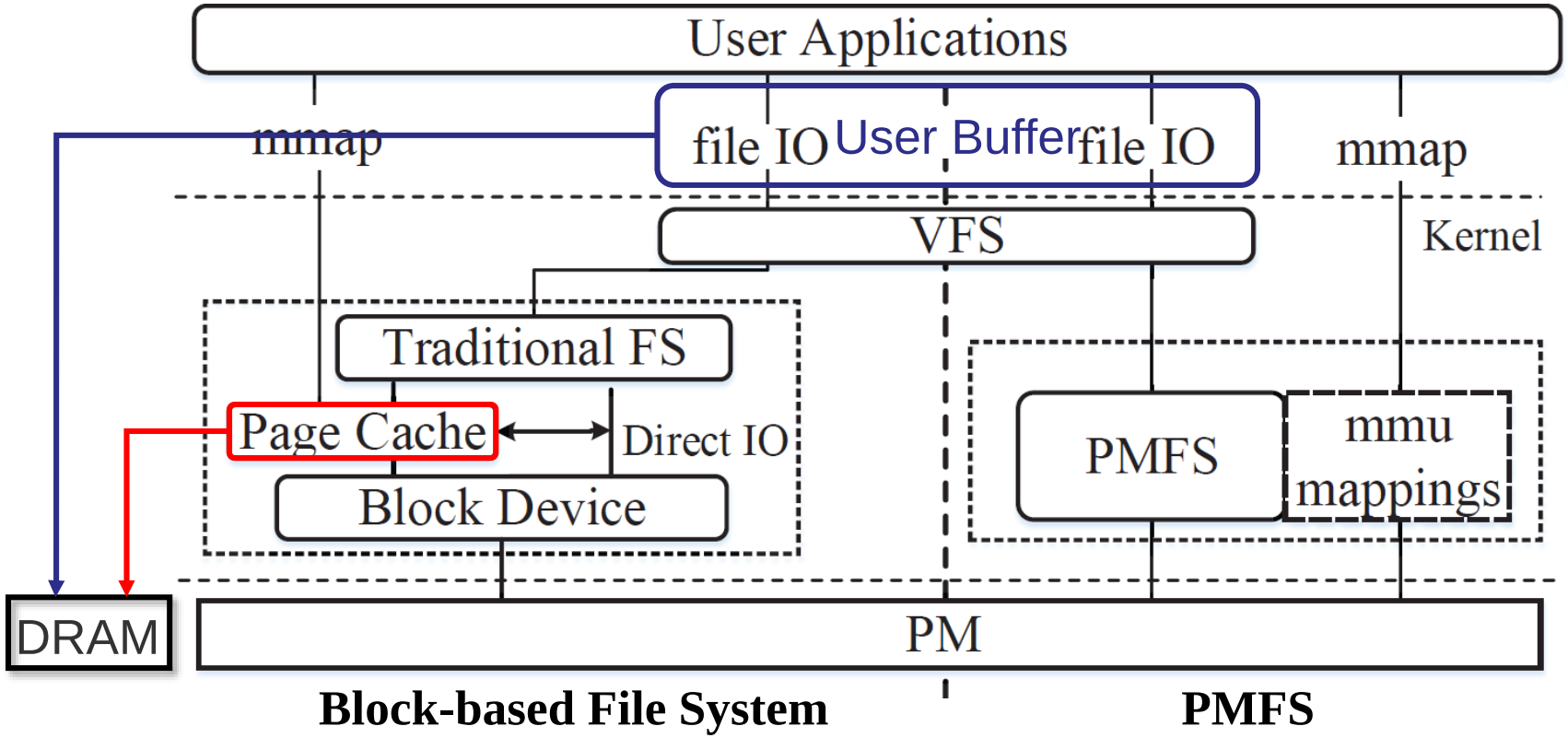
- Block Allocation: How to allocate disk space to files



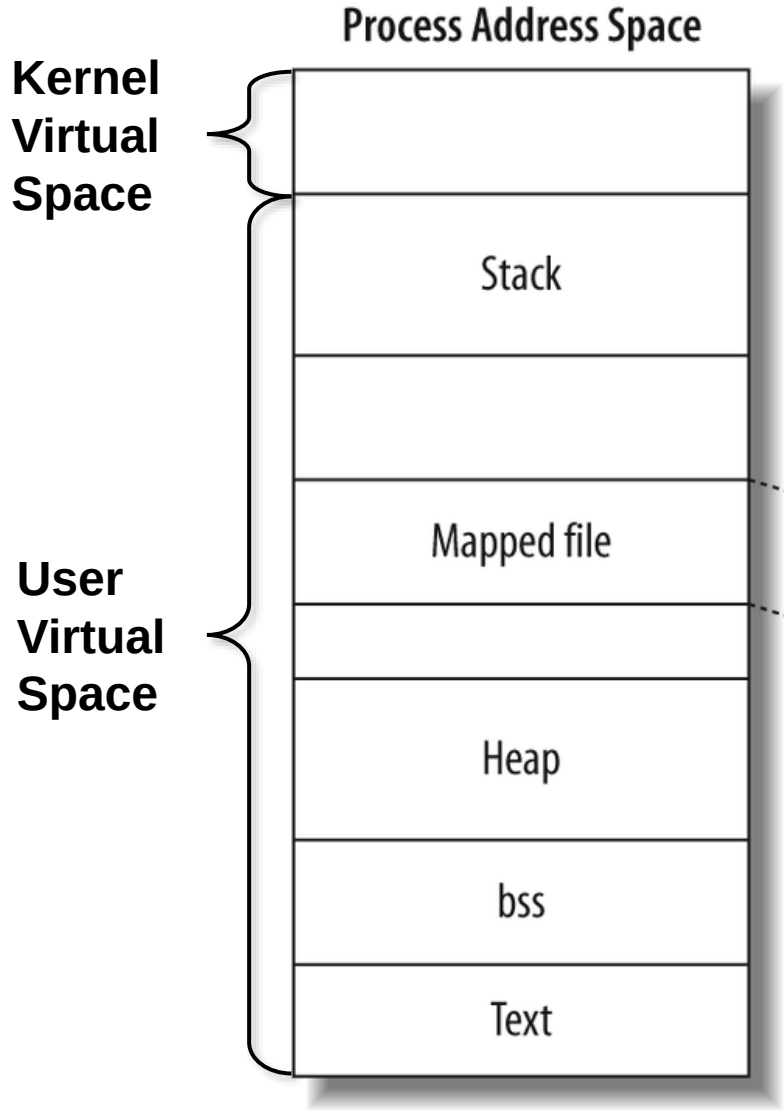
- It is a typical way to classify file system designs:
  - ① **Indexed Allocation**: an index block keeps block pointers
    - Examples: `UNIX FS`, `FFS`, `ext2`, `LFS`
  - ② **Linked Allocation:** each file is of linked blocks
    - Examples: `FAT`
  - ③ **Contiguous Allocation:** each file is of contiguous blocks
    - Examples: `ext4`

PM Optimizations: Memory-mapped I/O

- PMFS exploits **`mmap`** to map file data directly into the application's virtual address space.
  - Applications can access PM directly and efficiently without unnecessary copies (to DRAM) and software overheads.



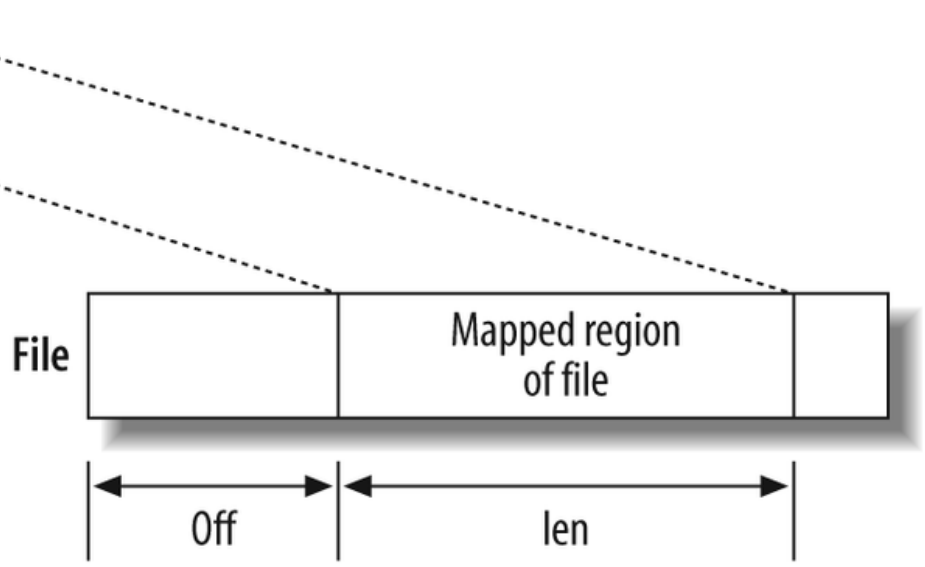
Review: Memory-mapped I/O (mmap)




```
void * mmap (void *addr, size_t len,
             int prot, int flags,
             int fd, off_t offset);
```

DESCRIPTION

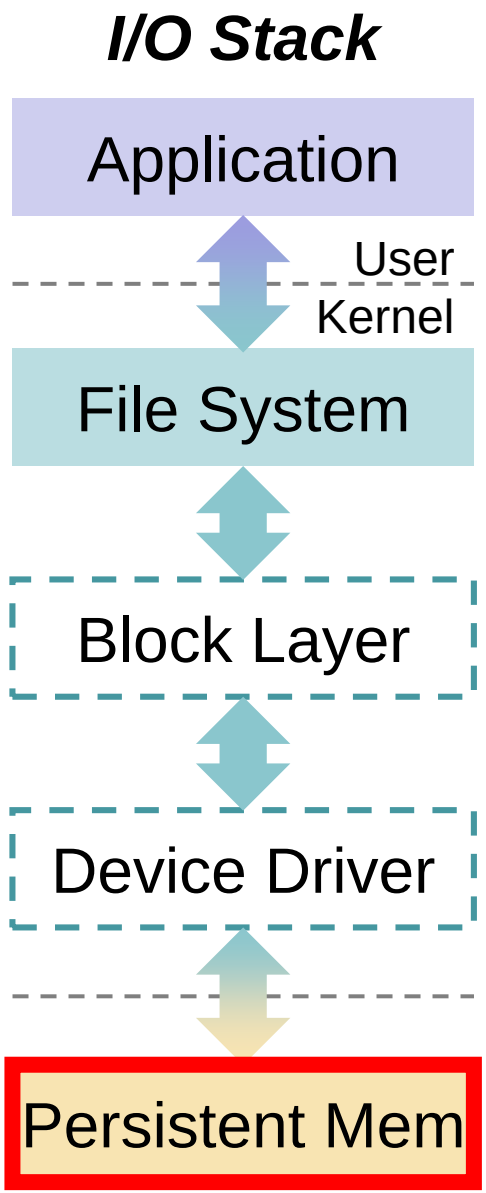
*`mmap()` creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in **`addr`**. The **`length`** argument specifies the length of the mapping. The **`prot`** argument describes the desired memory protection of the mapping.*



https://github.com/double-free/tiny-projects/tree/master/mmap
http://man7.org/linux/man-pages/man2/mmap.2.html

Outline

- Persistent Memory: Why and How
  - Emerging Persistent Memory Technologies
  - Characteristics and Integration Options
- Byte-addressable Persistent FS
  - System Architecture
  - Consistency: Short-Circuit Shadow Paging
  - Write Ordering: Epoch Barriers
- Persistent Memory FS
  - System Architecture
  - Optimizations for Byte-Addressable PM
  - Hybrid Approach for Consistency
  - Protection from Stray Writes
  - Write Ordering and Durability



Hybrid Approach for Consistency (1/2)

- PMFS uses a hybrid approach for consistency.
  - **Metadata Updates**: ① Atomic in-place updates and ② fine-grained logging for (usually small)
  - **File Data Updates**: ③ Page-based CoW
  - *Note: BPFS combines atomic in-place updates and CoW.*

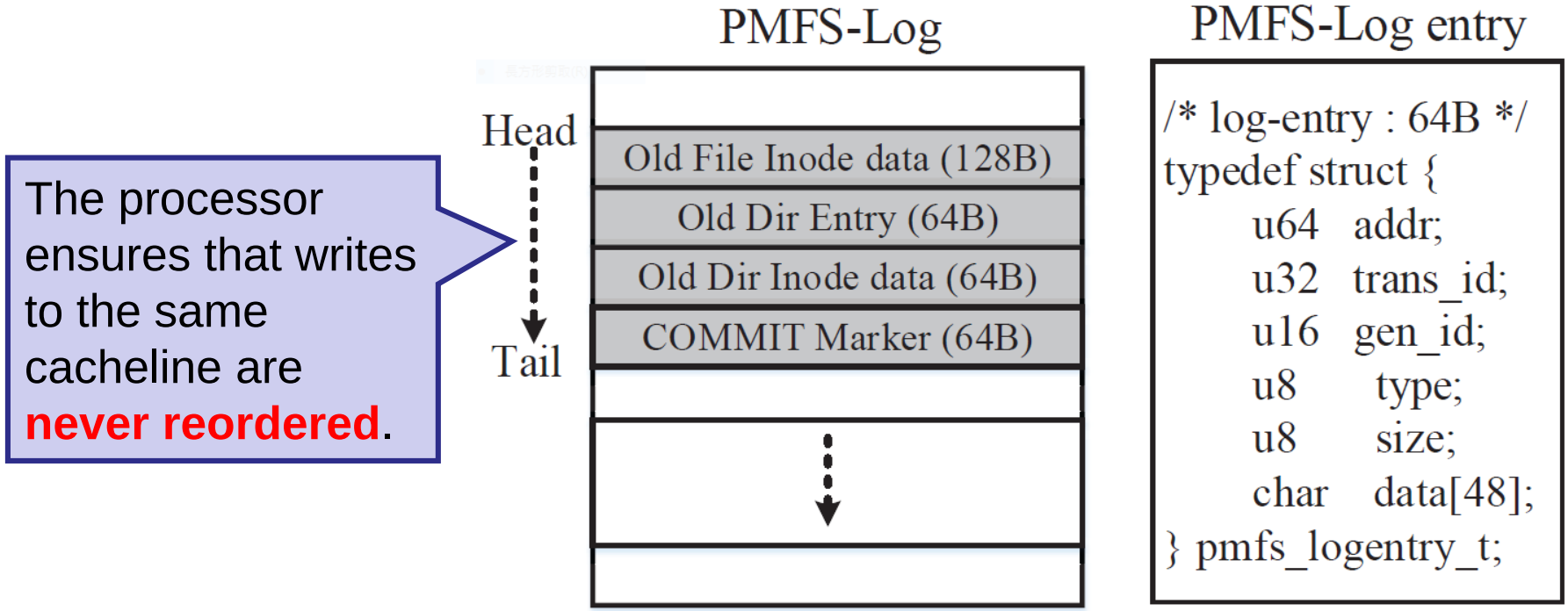
① Atomic In-place Update (for Metadata)

- PMFS leverages processor features for atomic in-place updates, avoiding logging in more cases:
  - **8-byte**: Natively supported by the processor.
    - *Note: BPFS only supports 8-byte atomic in-place update.*
  - **16-byte**: Supported via **`cmpxchg16b`** with **LOCK prefix**.
  - **64-byte** (i.e., **cacheline**): Supported if *Restricted Transactional Memory* (*RTM*) is available.

Hybrid Approach for Consistency (2/2)

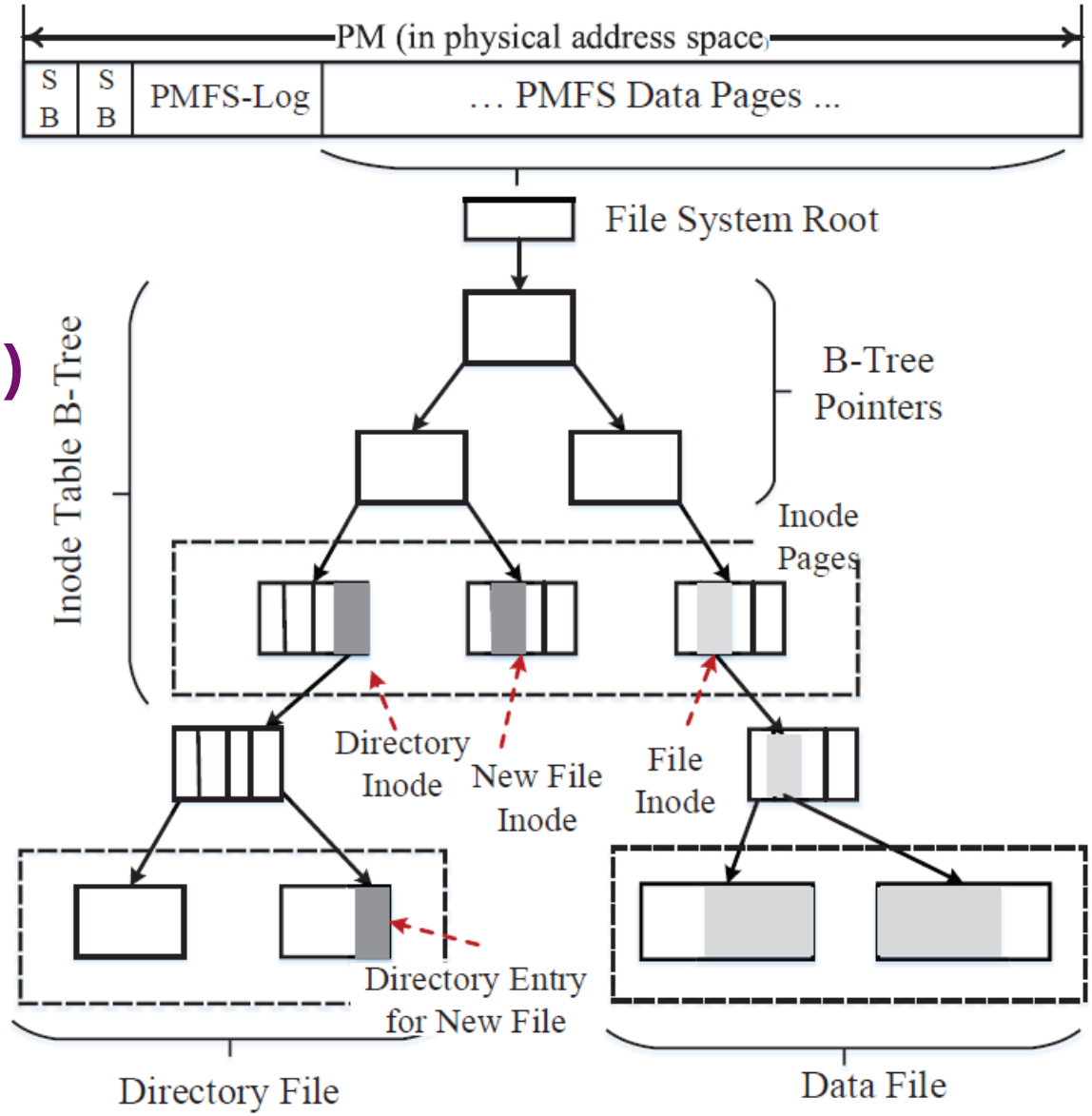
② Fine-grained Logging (for Metadata)

- PMFS logs metadata updates at cacheline granularity, reducing large write amplification caused by CoW.
  - The journal area, **`PMFS-Log`**, is a fixed-size circular buffer.
  - Each 64B log entry describes an update to the metadata.



Recall: PMFS Layout

- PM Address Space:
  - ① Superblock and its redundant copy
  - ② **A journal area (called PMFS-Log)**
  - ③ Dynamic allocated "pages"
- **B-trees** are used to organize the metadata:
  - ① Inode table
  - ② Directory inodes
  - ③ File inodes

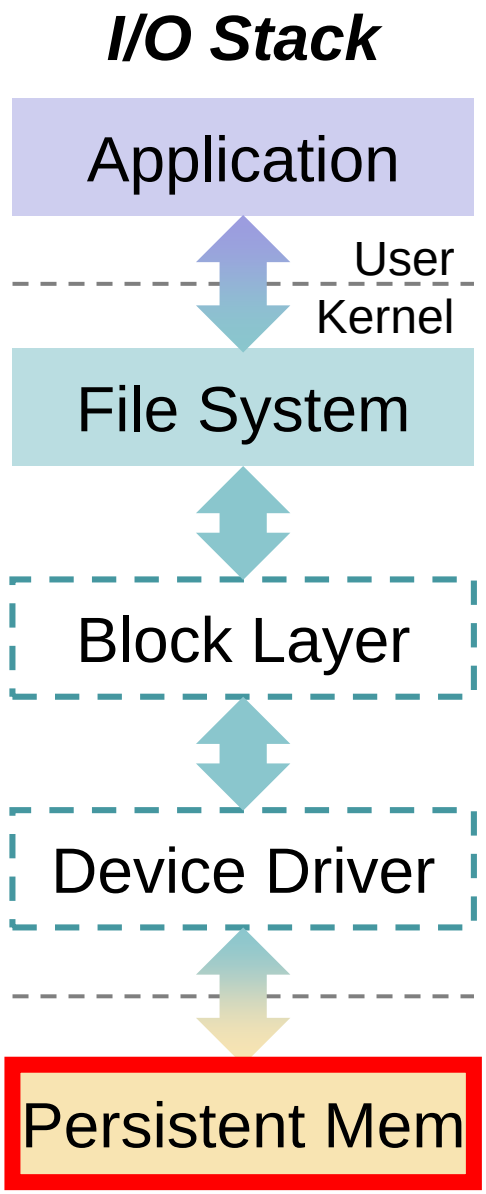


Outline

- Persistent Memory: Why and How
  - Emerging Persistent Memory Technologies
  - Characteristics and Integration Options
- Byte-addressable Persistent FS
  - System Architecture
  - Consistency: Short-Circuit Shadow Paging
  - Write Ordering: Epoch Barriers
- Persistent Memory FS
  - System Architecture
  - Optimizations for Byte-Addressable PM
  - Hybrid Approach for Consistency
  - Protection from Stray Writes
  - Write Ordering and Durability

Microsoft

intel



Protection from Stray Writes (1/2)

- PM is exposed to permanent corruption from "**stray writes**" due to bugs in the OS or drivers.
  - Why? PM can be directly accessed by memory-mapped I/O.
- PMFS protects PM from stray writes as follows:
  - **User-from-User**: by exploiting existing paging mechanism.
  - **Kernel-from-User**: by exploiting privilege levels.
  - **User-from-Kernel**: by disallowing kernel to access the user address space if SMAP is enabled.
  - **Kernel-from-Kernel**: by proposing a write window scheme.

	User Privilege	Kernel Privilege
User Address Space	*Process Isolation (via Paging)*	*Supervisor Mode Access Prevention (SMAP)*
Kernel Address Space	*Privilege Levels*	***Write Window***

Protection from Stray Writes (2/2)

- Write Window for "kernel-from-kernel" protection:

P: Read-only PM page in kernel virtual address
write(P): Write to page P in ring 0 (kernel)
GP: General protection fault

```
// CR0.WP in x86
   CR: Control Register
if (ring0 && CR0.WP == 0)
   write(P) is allowed;
else
   write(P) causes GP;
```

```
// Using CR0.WP in PMFS
disable_write_protection() {
   CR0.WP = 0;
   disable_interrupts();
}
enable_write_protection() {
   enable_interrupts();
   CR0.WP = 1;
}
```

```
// Writes to PM in PMFS
disable_write_protection();
write(P);
enable_write_protection();
```

Write Window

PMFS leverages the processor's **write protect control** (**`CR0.WP`**).

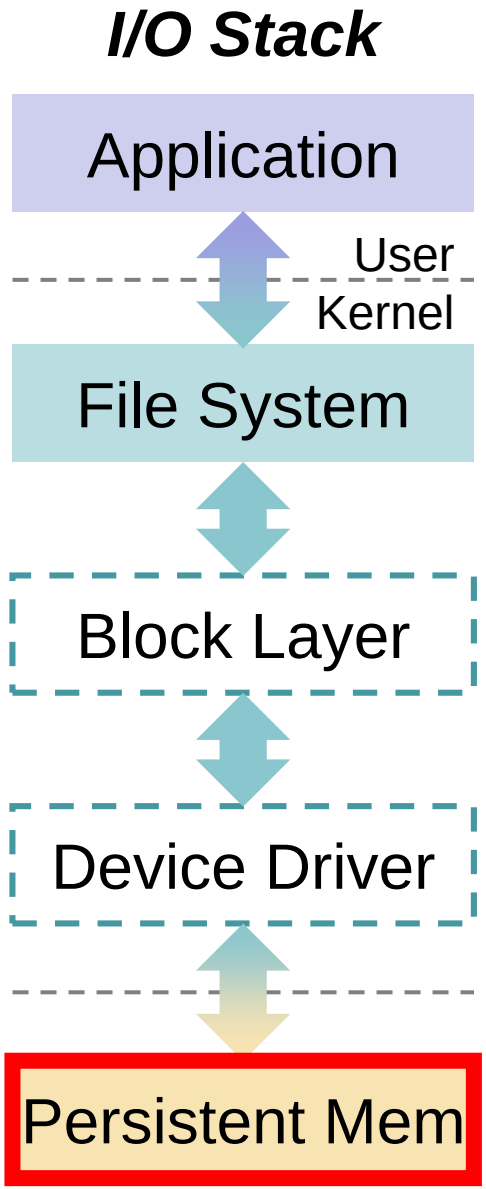
Interrupts must be disabled since CR0.WP is **not** saved across interrupts or context-switches.

Outline

- Persistent Memory: Why and How
  - Emerging Persistent Memory Technologies
  - Characteristics and Integration Options
- Byte-addressable Persistent FS
  - System Architecture
  - Consistency: Short-Circuit Shadow Paging
  - Write Ordering: Epoch Barriers
- Persistent Memory FS
  - System Architecture
  - Optimizations for Byte-Addressable PM
  - Hybrid Approach for Consistency
  - Protection from Stray Writes
  - Write Ordering and Durability

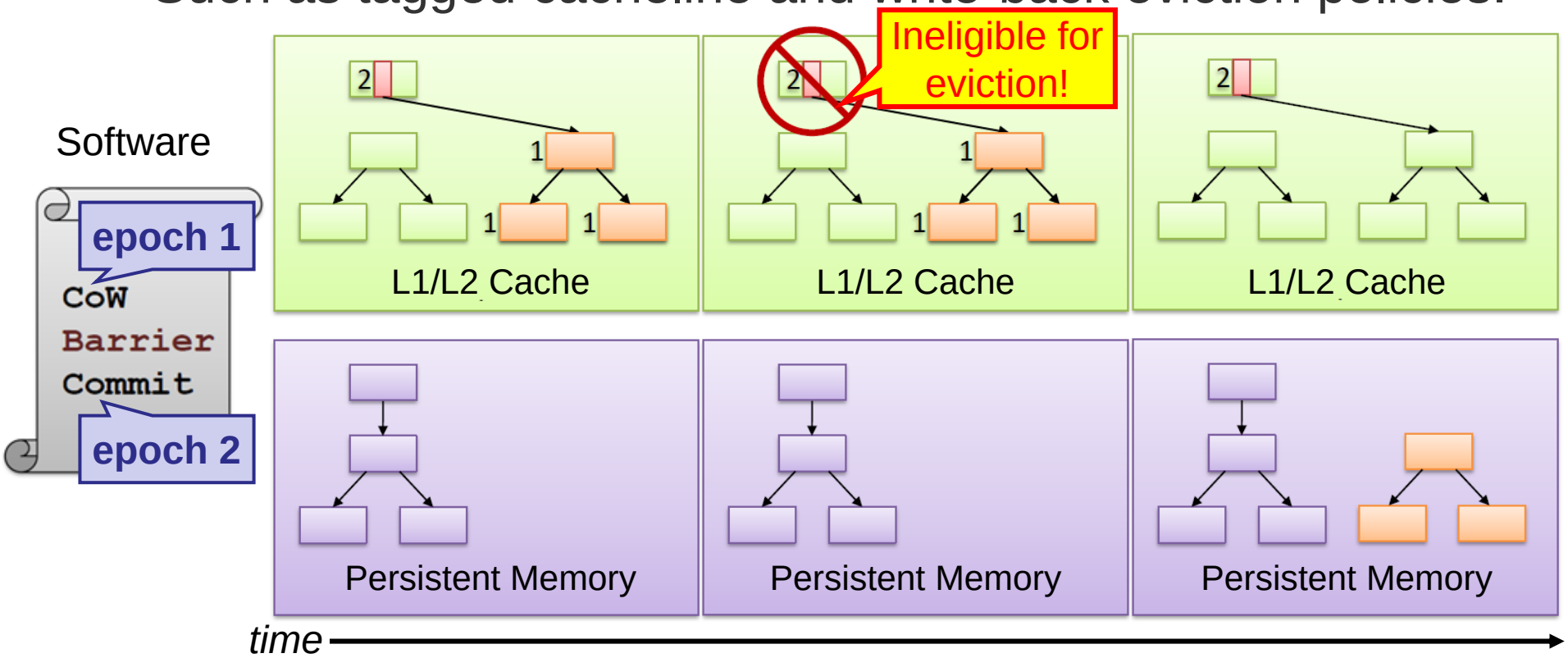
Microsoft

intel



Write Ordering and Durability (1/2)

- In BPFS, software simply issues epoch barriers and hardware guarantees the ordering within an epoch.
- However, an epoch-based solution would require non-trivial changes to cache and memory controllers.
  - Such as tagged cacheline and write-back eviction policies.



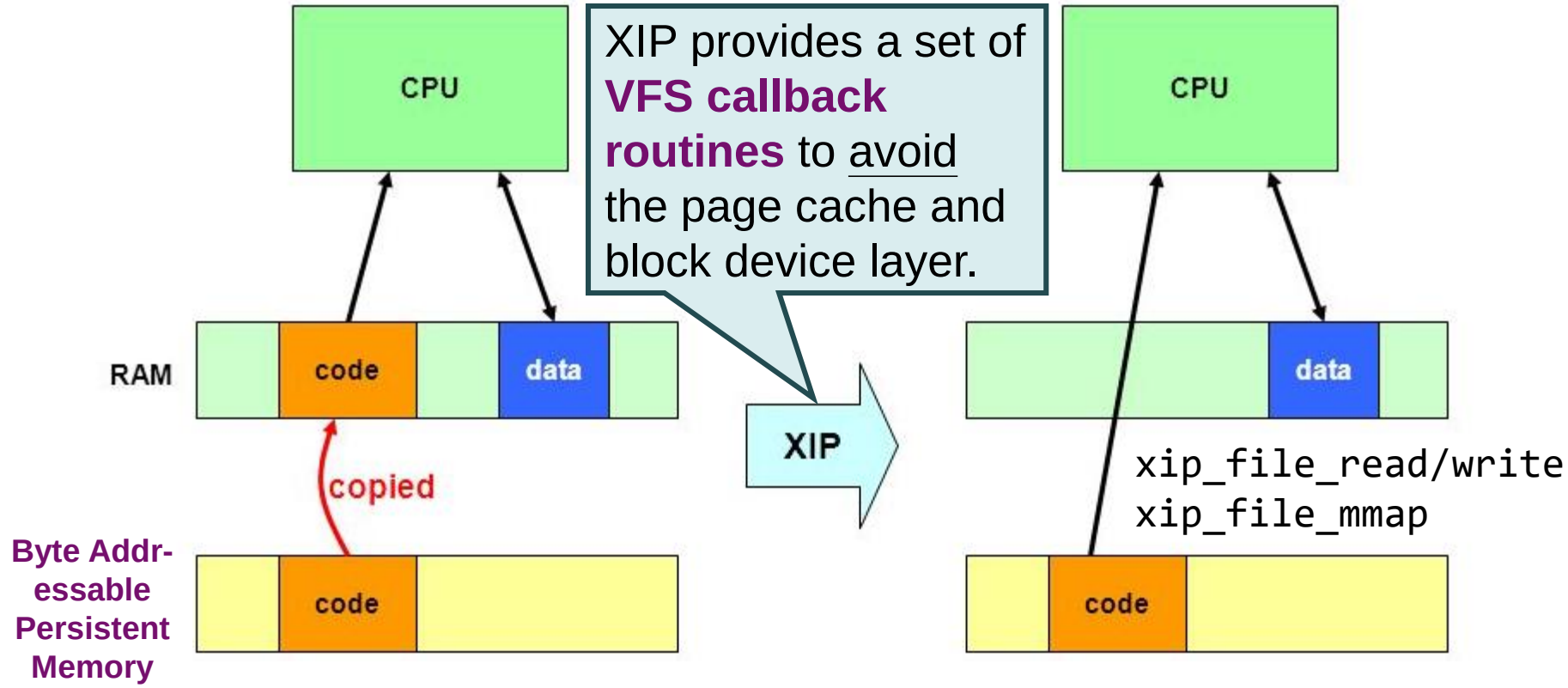
Write Ordering and Durability (2/2)

- Instead, PMFS ensures the write ordering and durability as follows:
  - ① PMFS enables software to **explicitly flush** modified data from volatile CPU caches.
    - The existing implementations of the `clflush` (cacheline flush) instruction are strongly ordered (with implicit fences), and may suffer from serious performance problems.
    - Therefore, PMFS revises the existing `clflush` instruction to provide improved performance through weaker ordering.
  - ② PMFS enforce the completion of cacheline flushing through **memory fences** (such as `sfence` instruction).
    - Reordering within a memory fence is possible, while ordering across the memory fence is strongly guaranteed.
  - ③ PMFS proposes a **hardware primitive** (**`pm_wbarrier`**) to guarantee the durability of the data written into PM.

Implementation Remarks (1/2)

- The prototype PMFS implementation is written as a Linux kernel module and has about **10K** lines of code.

① PMFS extends the **eXecute In Place (XIP)** interface in the Linux kernel.

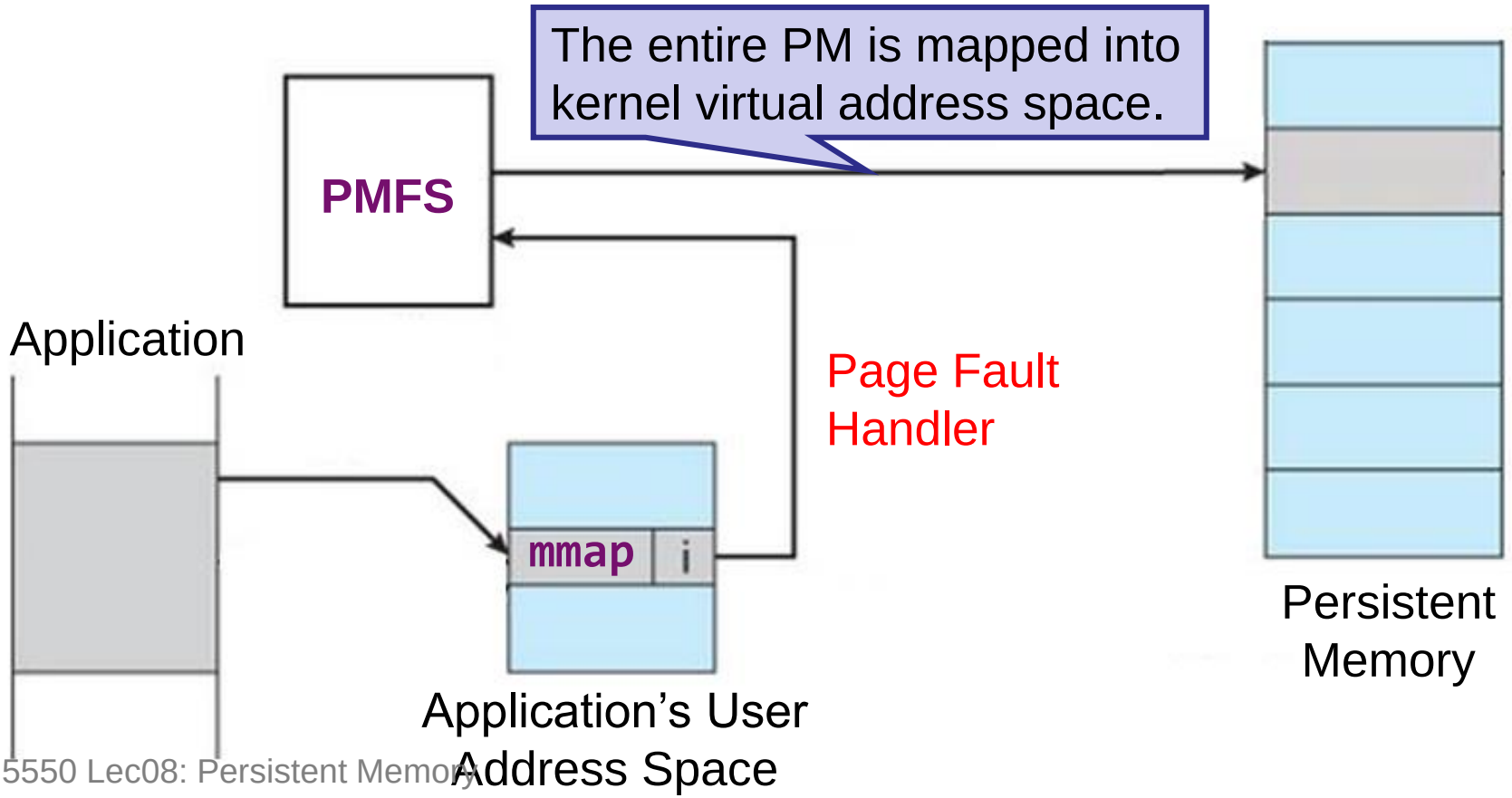


https://slideplayer.com/slide/10360595/

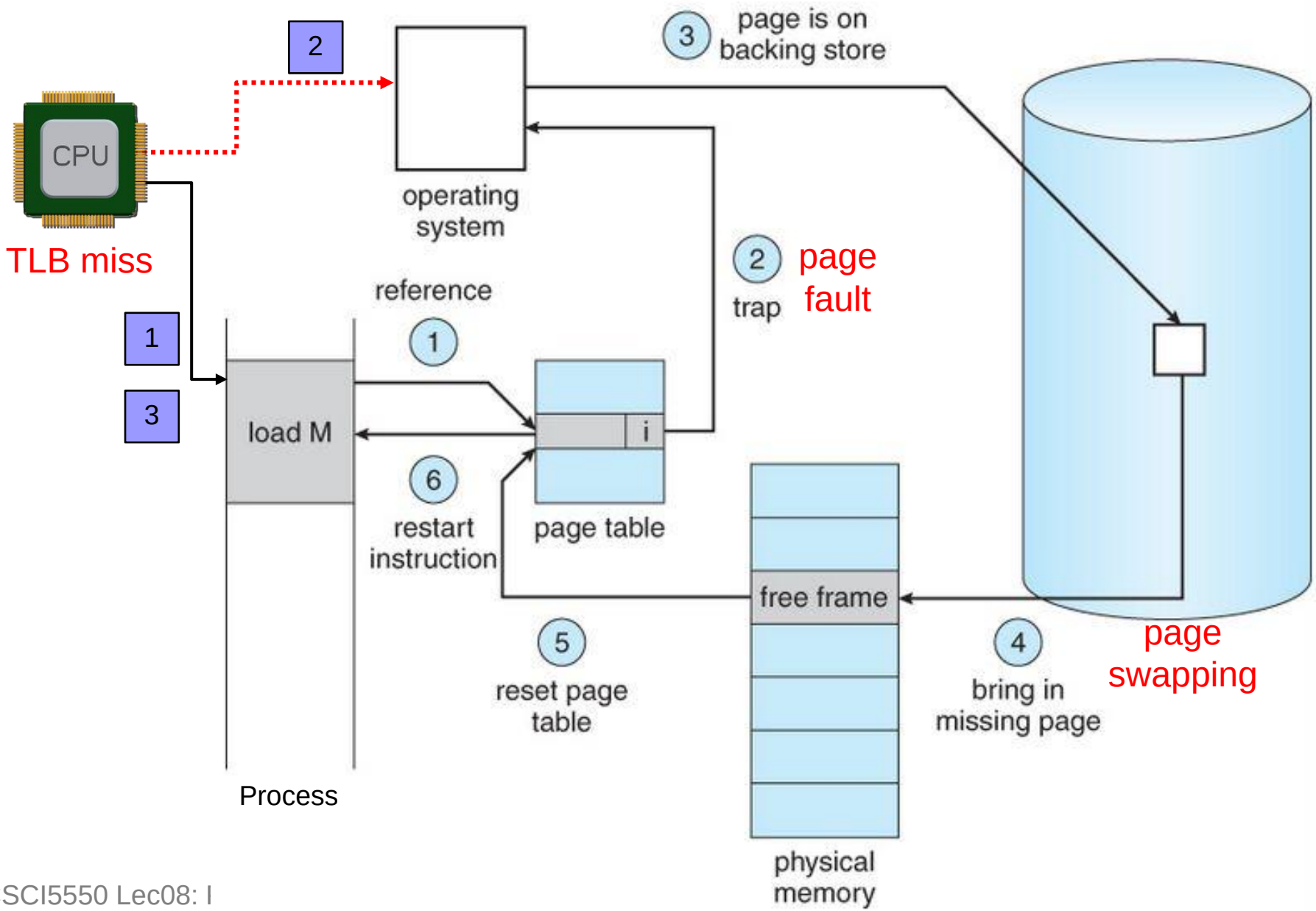
Implementation Remarks (2/2)

② To support direct mapping of PM, PMFS registers a page fault handler for the ranges in the application's address space that are backed by files in PMFS.

– This handler is invoked by the virtual memory subsystem.



Review: Page Fault Handling



Summary

- Persistent Memory: Why and How
  - Emerging Persistent Memory Technologies
  - Characteristics and Integration Options
- Byte-addressable Persistent FS
  - System Architecture
  - Consistency: Short-Circuit Shadow Paging
  - Write Ordering: Epoch Barriers
- Persistent Memory FS
  - System Architecture
  - Optimizations for Byte-Addressable PM
  - Hybrid Approach for Consistency
  - Protection from Stray Writes
  - Write Ordering and Durability

Microsoft

intel