Understanding Intel 2LM/PM in Linux

Alan Zhang

Mar. 28th, 2018

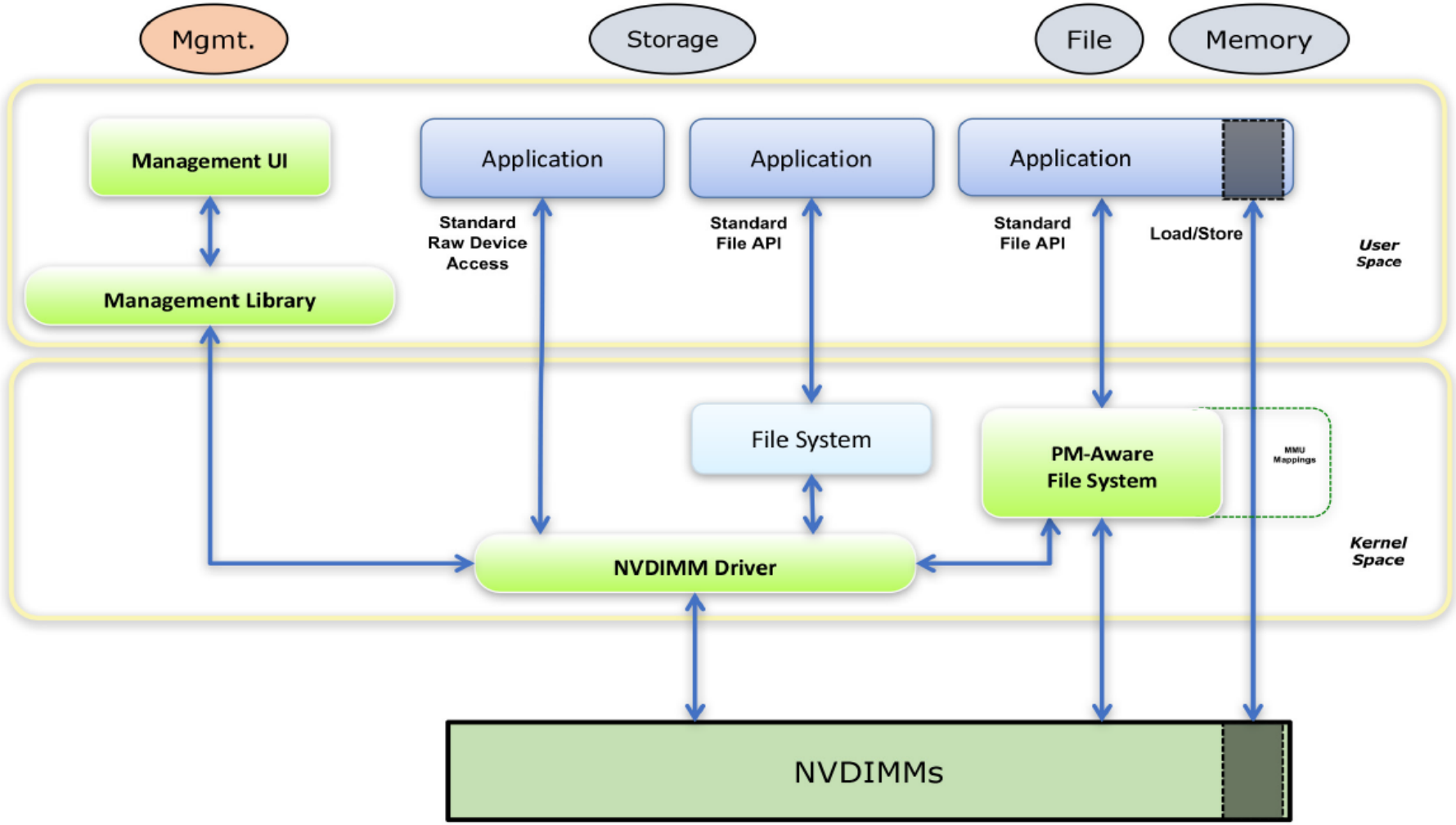
Agenda

- Concept of Persistent Memory
- Conceptual User Model of Persistent Memory in Linux
- Persistent Memory Usage at CPU Instruction Level
- Architecture Model of Persistent Memory in Linux
- Ecosystem and Industry Standards

Concept of Persistent Memory

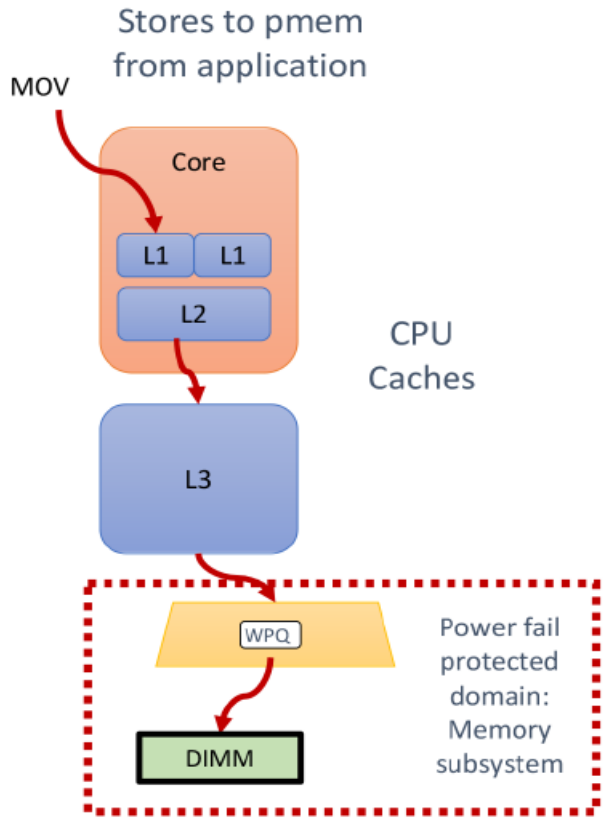
- Persistent memory (or storage class memory, the synonymous) is media w/ byte-addressable, load/store memory access, but w/ the persistence properties of storage. Narrow down PM in Intel solution, it is the NVDIMM that connects to system memory bus.
- Benefits as being operated in memory sematics
  - CPU cache coherency because NVDIMM is fast enough to stall CPU when an load/store instruction is accessing it
  - DMA
  - Byte addressability to support fine granularity manipulation

Persistent Memory User Model



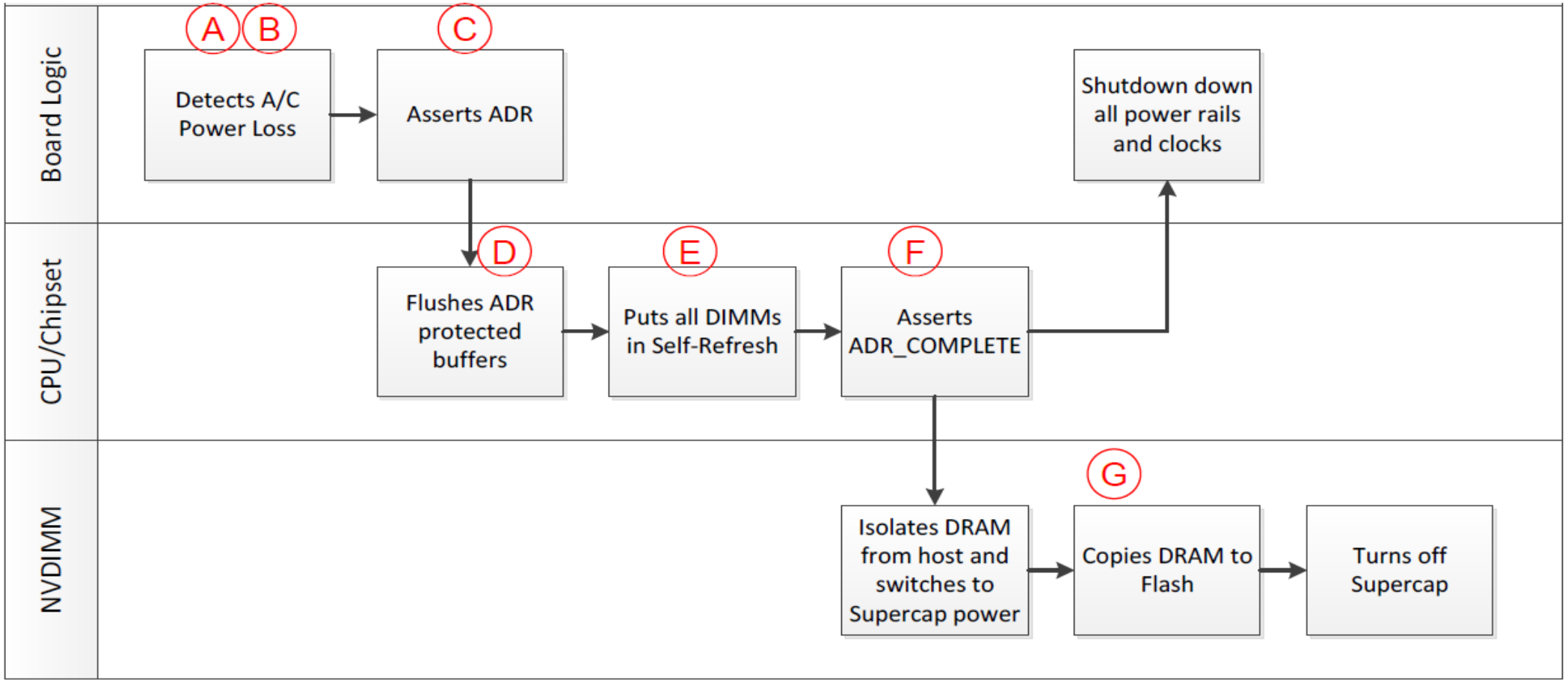
Persistent Memory in Runtime (1)

- Persistent memory aware file system works on DAX (Direct Access) that allows direct access to persistent memory w/o using the system page cache that is required for normal, storage-based files.
- Persistent memory can be mapped into memory using standard APIs like mmap() as shown in right picture. No kernel involvement, no interrupts for kernel/user-space context switch, only involve NVDIMM driver.
- To make runtime data persistent
  - From CPU cache to Persistent Memory via msync() or fsync() calls, flush CPU cache data into persistent memory
  - From Persistent Memory to traditional storage via ADR (Asynchronous DRAM refresh) which is supposed to be supported by NVDIMM



Persistent Memory in Runtime (2)

Asynchronous DRAM Re-fresh (ADR)



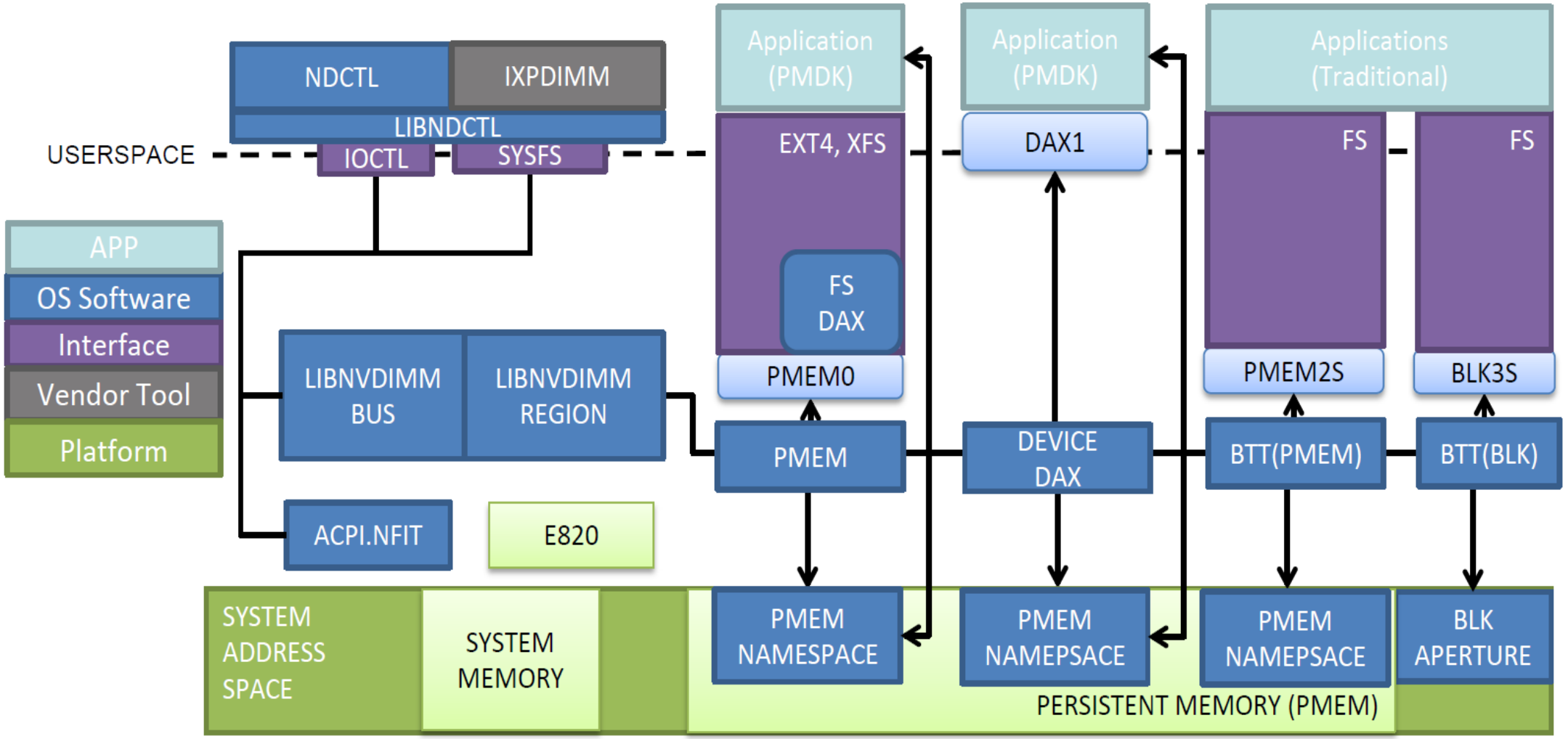
x86 Instructions for Transaction Operation on Persistent Memory

CLFLUSH	This instruction, supported in many generations of CPU, flushes a single cache line. Historically, this instruction is serialized, causing multiple CLFLUSH instructions to execute one after the other, without any concurrency.
CLFLUSHOPT (followed by an SFENCE)	This instruction, newly introduced for persistent memory support, is like CLFLUSH but without the serialization. To flush a range, software executes a CLFLUSHOPT instruction for each 64-byte cache line in the range, followed by a single SFENCE instruction to ensure the flushes are complete before continuing. CLFLUSHOPT is optimized (hence the name) to allow some concurrency when executing multiple CLFLUSHOPT instructions back-to-back.
CLWB (followed by an SFENCE)	Another newly introduced instruction, CLWB stands for *cache line write back*. The effect is the same as CLFLUSHOPT except that the cache line may remain valid in the cache (but no longer dirty, since it was flushed). This makes it more likely to get a cache hit on this line as the data is accessed again later.
NT stores (followed by an SFENCE)	Another feature that has been around for a while in x86 CPUs is the non-temporal store. These stores are "write combining" and bypass the CPU cache, so using them does not require a flush. The final SFENCE instruction is still required to ensure the stores have reached the persistence domain.
WBINVD	This kernel-mode-only instruction flushes and invalidates every cache line on the CPU that executes it. After executing this on all CPUs, all stores to persistent memory are certainly in the persistence domain, but all cache lines are empty, impacting performance. In addition, the overhead of sending a message to each CPU to execute this instruction can be significant. Because of this, WBINVD is only expected to be used by the kernel for flushing very large ranges, many megabytes at least.

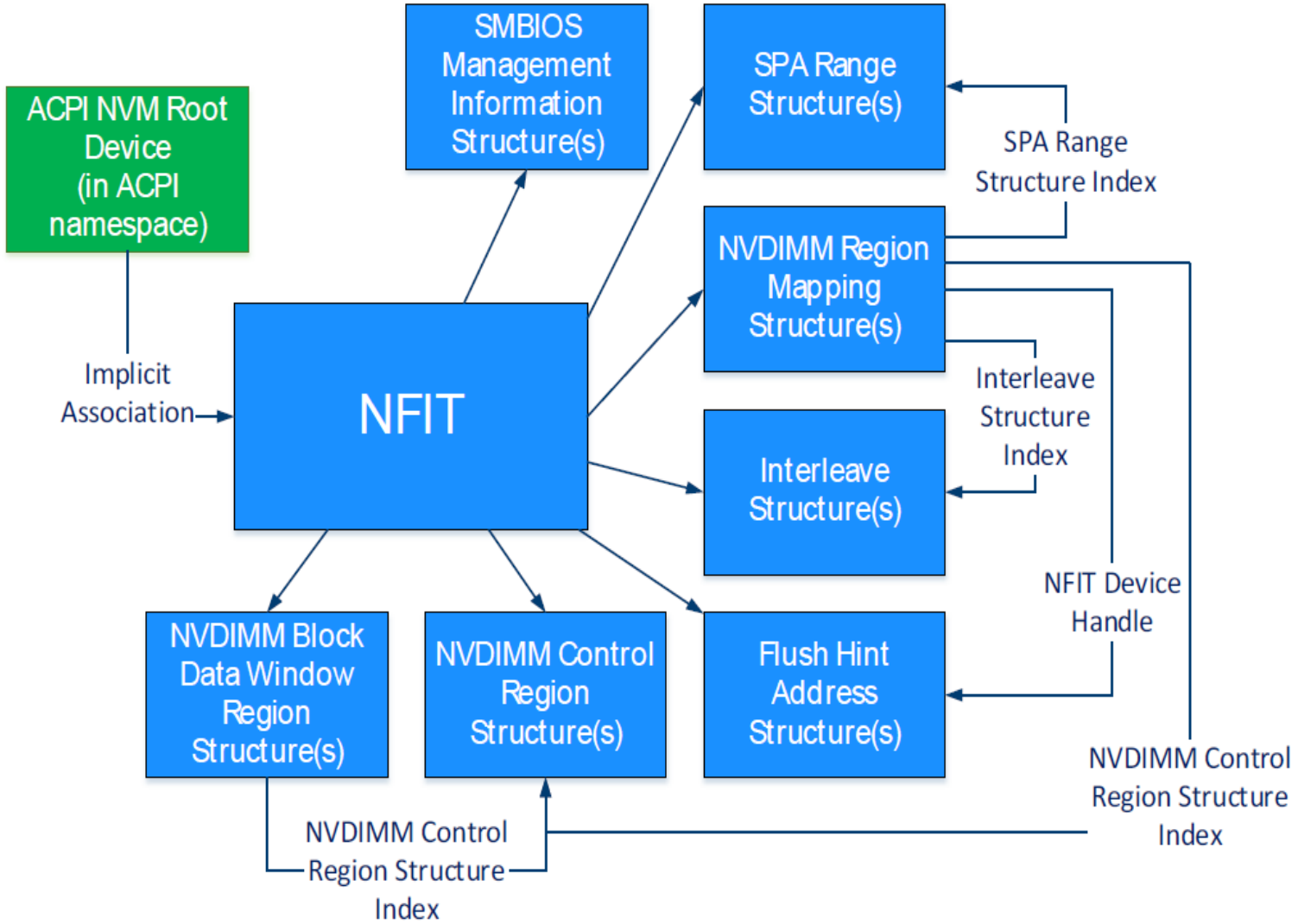
Challenge of Using Persistent Memory by Applications

- Transaction atomicity
  - Not allow the action of flushing persistent memory to be interrupted by other threads recall.
  - Transaction are power fail safe: Intel platform builds the eight-bytes power-fail-atomic store by hardware, so for application developers, they MUST reference the interfaces defined in Persistent Memory Programming Model.
- Managing Persistent Memory Space
  - Support both byte addressable memory and block device models
- Naming, Permission, and Isolation the usage of persistent memory by user-space applications

Persistent Memory – Architecture in Linux



NVDIMM Firmware Interface Table (NFIT)



http://www.uefi.org/sites/default/files/resources/ACPI_6.0.pdf

E820 Table Example

E820 is shorthand to refer to the facility by which the BIOS of x86-based computer systems reports the memory map to the operating system or boot loader.

```
[root@localhost Desktop]# dmesg |grep e820
 BIOS-e820: 0000000000000000 - 000000000009ac00 (usable)
 BIOS-e820: 000000000009ac00 - 00000000000a0000 (reserved)
 BIOS-e820: 00000000000e0000 - 0000000000100000 (reserved)
 BIOS-e820: 0000000000100000 - 000000007d4a1000 (usable)
 BIOS-e820: 000000007d4a1000 - 000000007d4e0000 (reserved)
 BIOS-e820: 000000007d4e0000 - 000000007d5f6000 (ACPI data)
 BIOS-e820: 000000007d5f6000 - 000000007e1ff000 (ACPI NVS)
 BIOS-e820: 000000007e1ff000 - 000000007f271000 (reserved)
 BIOS-e820: 000000007f271000 - 000000007f272000 (usable)
 BIOS-e820: 000000007f272000 - 000000007f2f8000 (ACPI NVS)
 BIOS-e820: 000000007f2f8000 - 000000007f800000 (usable)
 BIOS-e820: 0000000080000000 - 0000000090000000 (reserved)
 BIOS-e820: 00000000fed1c000 - 00000000fed20000 (reserved)
 BIOS-e820: 00000000ff000000 - 0000000100000000 (reserved)
 BIOS-e820: 0000000100000000 - 0000000200000000 type 12
e820 update range: 0000000000000000 - 0000000000010000 (usable) ==> (reserved)
e820 update range: 0000000000000000 - 0000000000001000 (usable) ==> (reserved)
e820 remove range: 00000000000a0000 - 0000000000100000 (usable)
e820 update range: 0000000080000000 - 0000000100000000 (usable) ==> (reserved)
```

the nvdimm memory address arrange in e820 map

Note: ACPI 6.0 defines Type 7 for Persistent Memory and NFIT

NVDIMM Root Device DSMs

Query Command Implemented
Query Address Range Scrub Capabilities
Start Address Range Scrub
Query Address Range Scrub Status
Clear Uncorrectable Error
Translate SPA
Reserved
ARS Error Inject
ARS Error Inject Clear
ARS Error Inject Status Query

http://www.uefi.org/sites/default/files/resources/ACPI_6.0.pdf

Example for DSM

NVDIMM IFC 0x0201/0x0301* DSMs

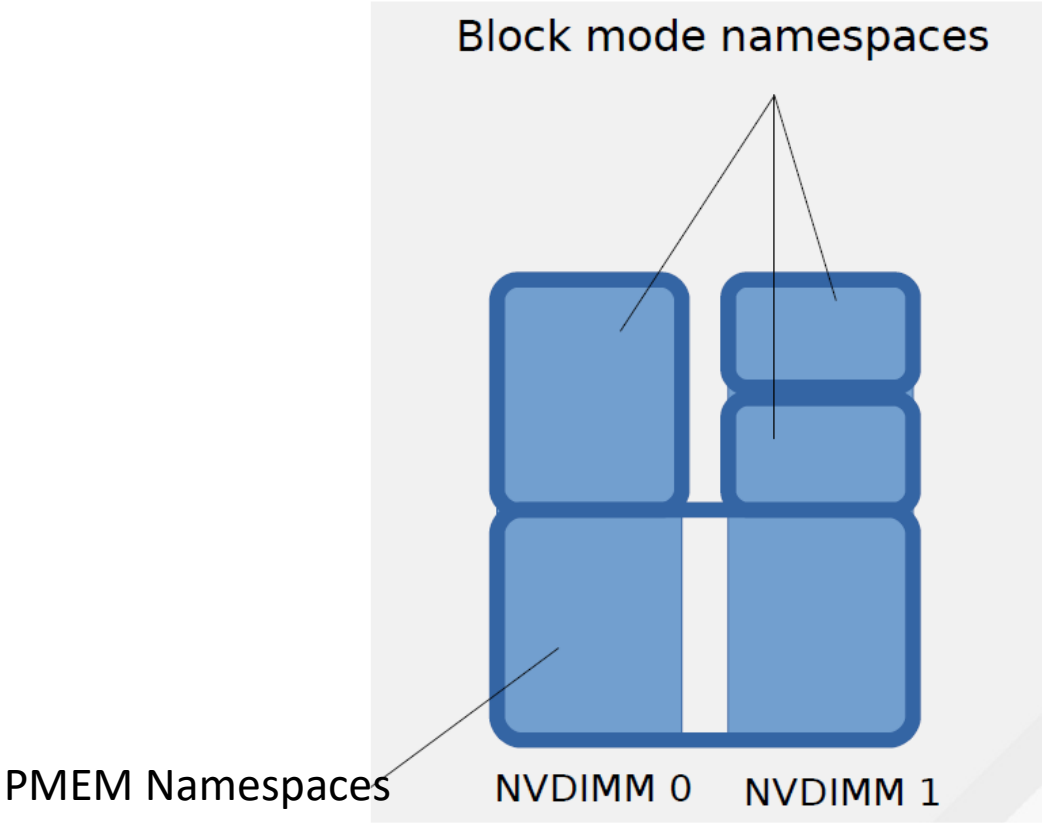
Query Command Implemented	Get Supported Modes
Get SMART and Health Info	Get FW Info
Get SMART Threshold	Start FW Update
Get Block NVDIMM Flags	Send FW Update Data
Get Command Effect Log Size	Finish FW Update
Get Command Effect Log	Query Finish FW Update Status
Pass-Through Command	Set SMART Threshold
Enable Latch System Shutdown Status	Inject Error

LIBNVDIMM Kernel Device Model and LIBNDCTL Userspace API (1)

- Three device drivers corresponding to enumerated NFIT namespaces' definition
  - nd_pmem: /dev/pmem → PMEM namespace defined persistent memory range
  - nd_blk: /dev/nd_blk → BLK namespace defined memory-mapped-I/O apertures to access persistent storage
  - nd_btt: /dev/btt → BTT namespace defined physical memory device in memory disk semantics (atomic sector update)
- Support PMEM and BLK access modes on NVDIMMs via defining PMEM namespace, and BLK namespace
- PMEM namespace behavior
  - Access using load and store
  - Mapped to system physical memory
- BLK namespace behavior
  - Access using block operation
  - Atomicity at block level: in case of power failure when writing, it can be rolled back/forward
  - Indirect access through a BLOCK WINDOW: DSM example definition for BLK mode
    - No mapping the entire memory
    - Reduce address utilization
    - Reduce risk of wrong addressed writes

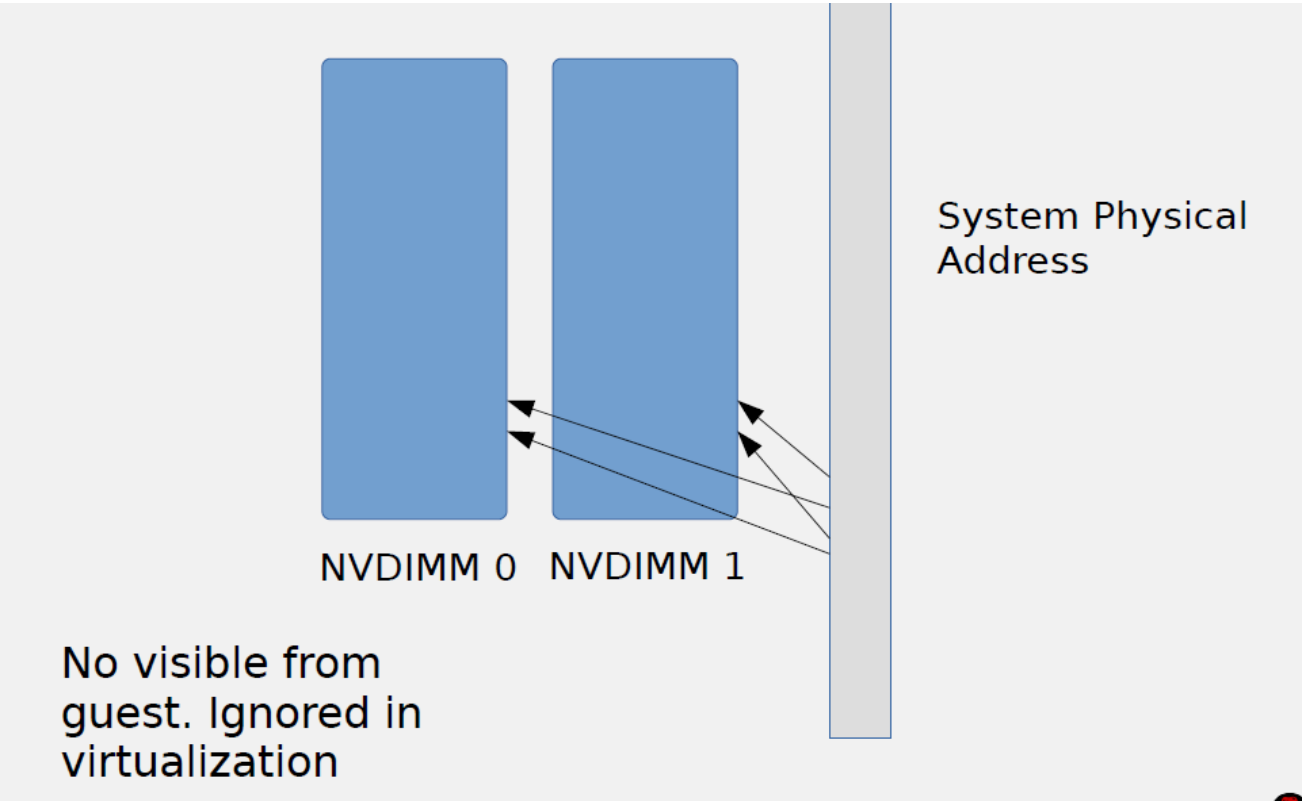
LIBNVDIMM Kernel Device Model and LIBNDCTL Userspace API (2)

- Namespaces of PMEM and BLK defined in NFIT CANNOT be overlap
- PMEM namespaces can span across multiple NVDIMMs (interleaving)
- BLK mode namespaces can be in just one NVDIMM, there can be multiple BLK mode namespaces in one NVDIMM. For control purpose, there defined BLK Window namespace that convers BLK Control Window, and BLK Data Window.



LIBNVDIMM Kernel Device Model and LIBNDCTL Userspace API (3)

- PMEM namespaces support interleaving



LIBNVDIMM Kernel Device Model and LIBNDCTL Userspace API (4)

- BTT (Block Translation Table) namespace (Optional)
  - Atomicity at block level in block namespaces
  - Block namespaces broken into arenas (up to 512GB)
    - Arena info block (backup)
    - Arena flog
    - Arena map
    - Arena data area
    - Arena info block

LIBNVDIMM Kernel Device Model and LIBNDCTL Userspace API (5)

- LIBNVDIMM sysfs layout
  - Control class device in /sys/class/nd/ndctlX : accept DSM messages to be passed to DIMM identified by its NFIT handle.
  - Bus : sysfs “Bus” corresponding to NFIT configuration, 1:1 relation. /sys/devices/platform/nfit_XXXX.0/ndbusY
  - DIMM: /sys/devices/platform/nfit_XXXX.0/ndbusY/nmemZ
  - Region: /sys/devices/platform/nfit_XXXX.0/ndbusY/regionZ
- LIBNDCTL APIs
  - Context: ndctl_new(&ctx)
  - DSM interface in sysfs: /sys/class/nd/ndctlX
  - Bus Operation APIs:
    - Registration: struct nvdimm_bus * nvdimm_bus_register(struct device * parent, structure nvdimm_bus_descriptor * nfit_desc)
    - Enumeration: ndctl_bus_foreach(ctx, bus)
  - DIMM APIs
    - Create: struct nvdimm *nvdimm_create(struct nvdimm_bus *nvdimm_bus, ……)
    - Enumeration: ndctl_dimm_foreach()
  - Region: Registered for each PMEM range or BLK-aperture set, and the APIs respectively
    - struct nd_region *nvdimm_pmem_region_create(struct nvdimm_bus *nvdimm_bus, ……)
    - struct nd_region *nvdimm_blk_region_create(struct nvdimm_bus *nvdimm_bus, ……);

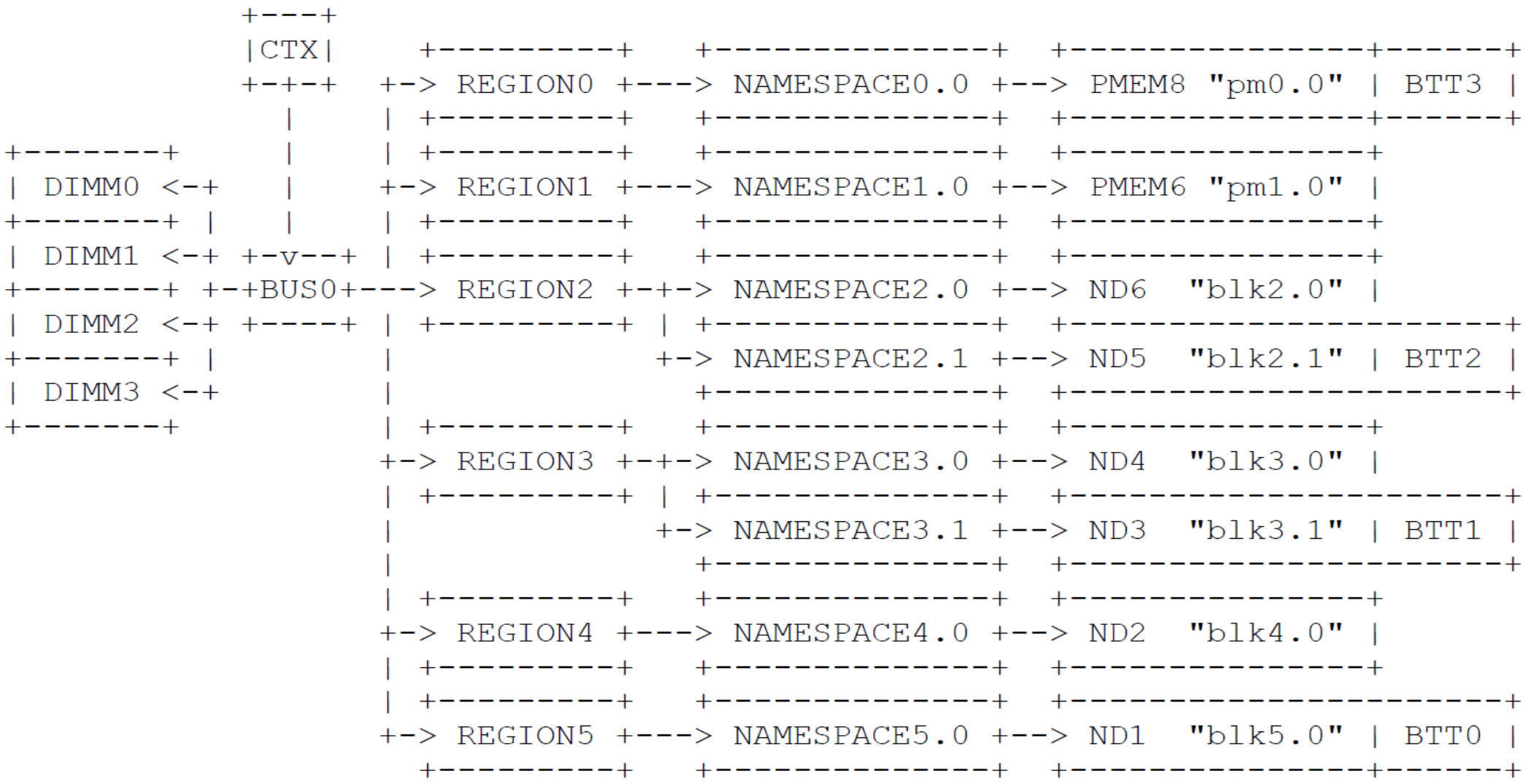
LIBNVDIMM Kernel Device Model and LIBNDCTL Userspace API (6)

- LIBNDCTL APIs
  - Namespace: resolving DPA aliasing and LABEL specified boundaries in REGION, and populates one or more “namespace” devices. The arrival of “namespace” devices trigger either nd_blk or nd_pmem driver to be loaded and register a disk/block device. Namespaces are indexed relative to their parent region.
  - For BLK namespace device, it has “sector_size” attribute;
  - Sysfs node location: /sys/devices/platform/nfit_XXX.Y/ndbusZ/namespaceM.N
  - Namespace APIs
    - Namespace creation: Idle namespaces are automatically created by kernel if a given region has enough available capacity to create a new namespace.
      - ndctl_namespace_set_alt_name(ndns, devname)
      - ndctl_namespace_set_uuid(ndns, parameters->uuid)
      - ndctl_namespace_set_size(ndns, parameters->size)
      - ndctl_namespace_set_sector_size(ndns, parameters->lbasize) if BLK namespace
      - ndctl_namespace_enable(ndns)
    - Namespace enumeration: ndctl_namespace_foreach(region, ndns)

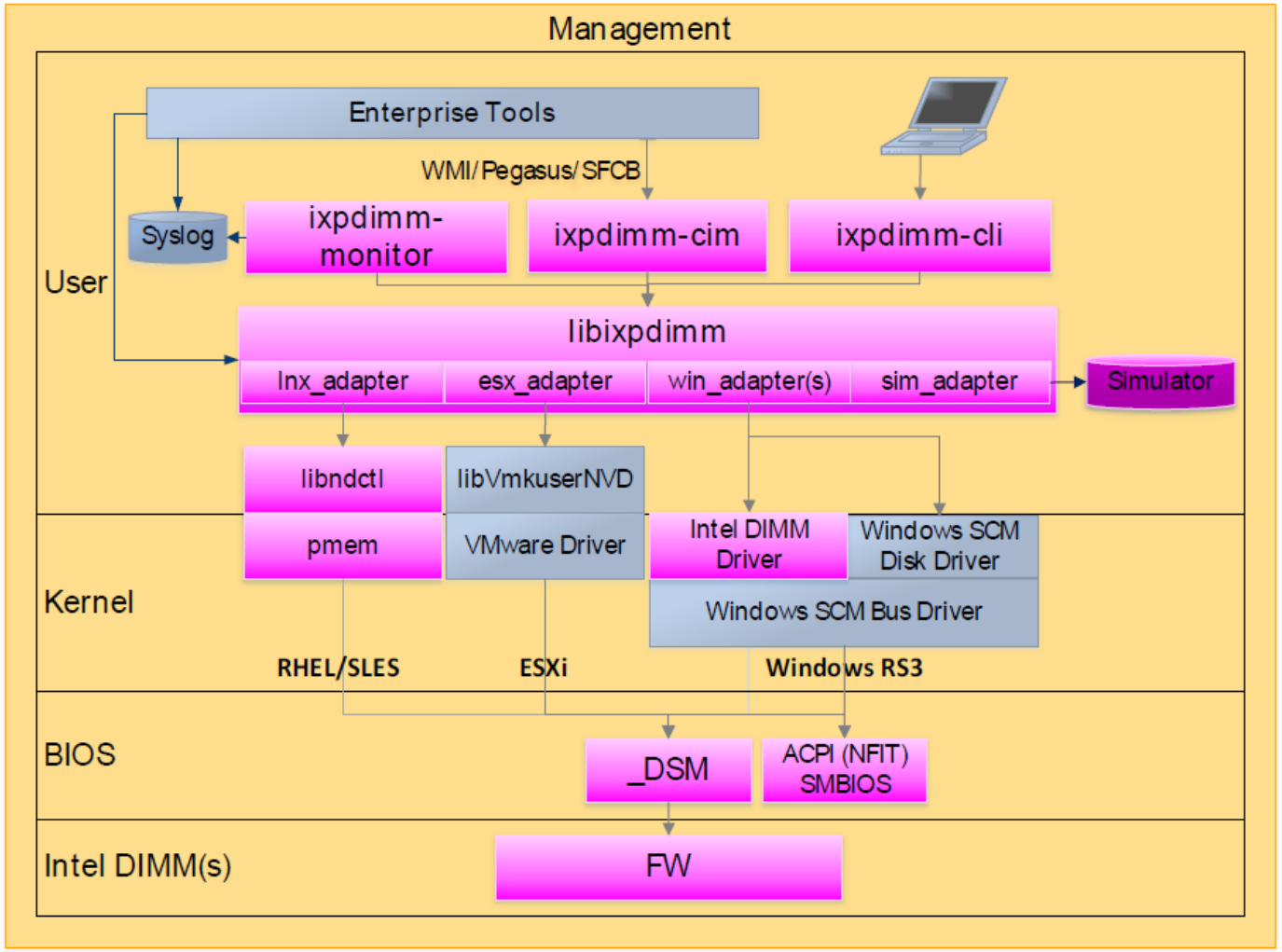
LIBNVDIMM Kernel Device Model and LIBNDCTL Userspace API (7)

- LIBNDCTL APIs
  - Block Translation Table (BTT): block device driver that fronts either the whole block device or a partition of a block device emitted by either a PMEM or BLK namespace. Every region starts with at least one BTT (seed device) w/ attributes “namespace”, “uuid”, and “sector_size”, and then bind the BTT device to the nd_pmem or nd_blk driver depend on the region type.
  - Sysfs node location: /sys/devices/platform/nfit_XXX.Y/ndbusZ/regionM/bttN
  - BTT creation: An idle BTT device is created automatically per region, which is handled as “seed” btt device, and then assign it to consume PMEM or BLK namespace.
    - ndctl_btt_foreach(region, btt)
    - ndctl_btt_set_uuid(btt, parameters->uuid)
    - ndctrl_btt_set_sector_size(btt, parameters->sector_size)
    - ndctl_btt_set_namespace(btt, parameters->ndns)
    - ndctrl_btt_enable(btt)

Concepts of NVDIMM Device Model in Linux



IXPDIMM – Intel NVDIMM TOOL



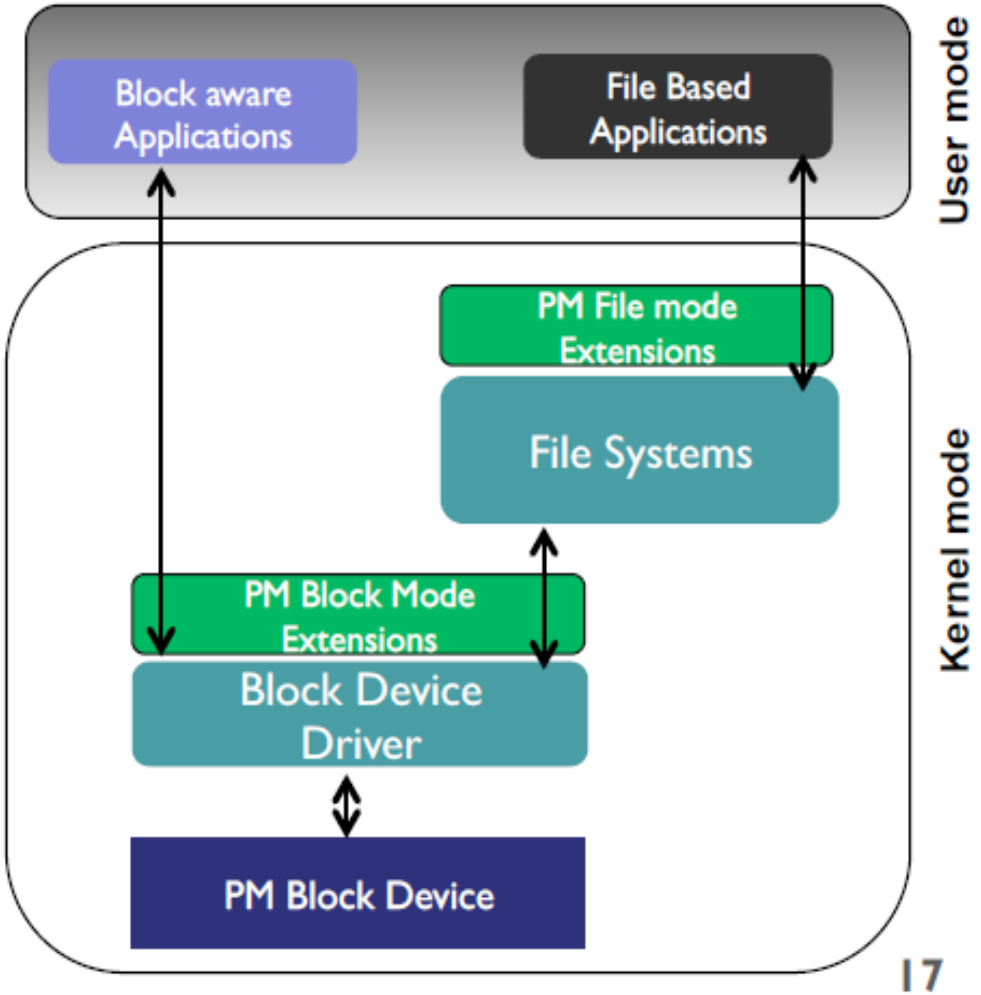
Package	Repository
ixpdimm sw	https://github.com/01org/ixpdimm_sw
nvm frameworks	https://github.com/01org/invm-frameworks

Persistent Memory Development Kit

- PMDK which allows applications to access persistent memory as *memory-mapped files*, as described in the SNIA NVM Programming Model.
- PMDK is a project with the goal of making persistent memory programming easier. It currently supports **ten libraries**, targeted at various use cases for persistent memory, along with language support for **C**, **C++**, **Java**, and **Python**, tools like the **pmemcheck** plug-in for valgrind, and an increasing body of documentation, code examples, tutorials, and blog entries.
- Source for PMDK: https://github.com/pmem/pmdk/

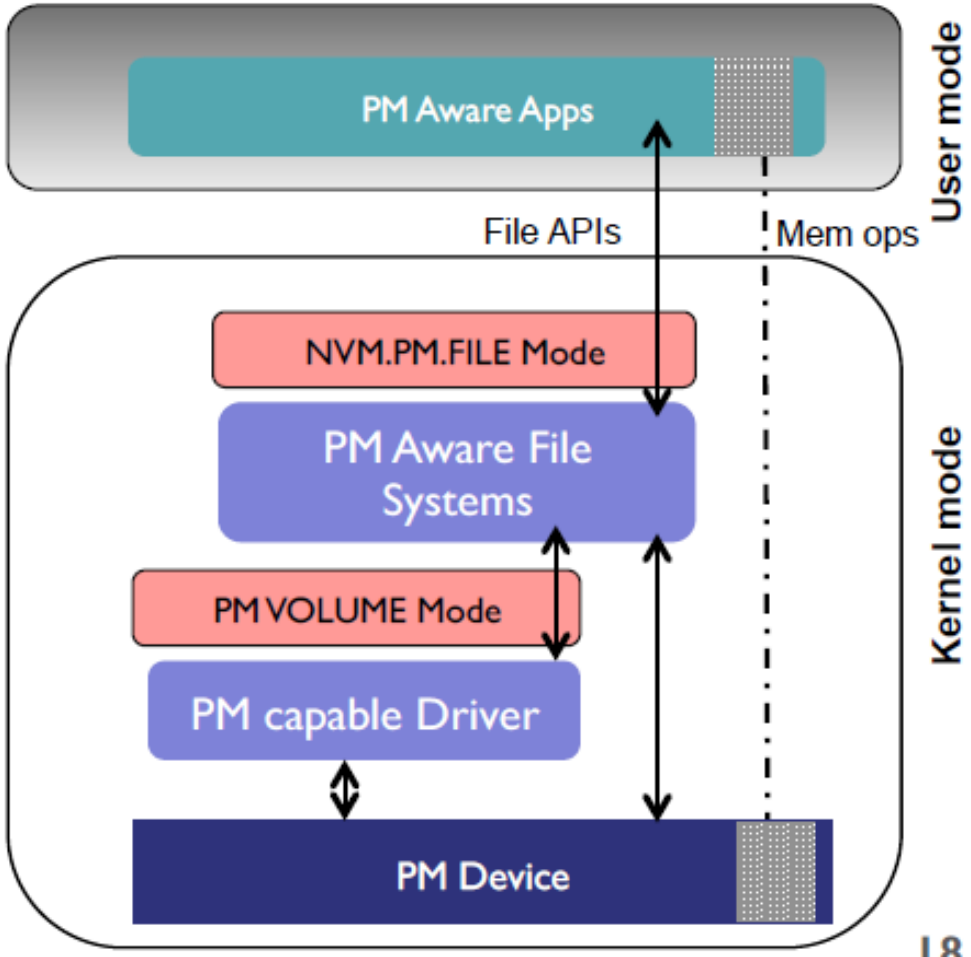
Programming Models Supported in PMDK (1)

- NVM.BLOCK Mode
  - Targeted for file systems and block-aware applications
  - Atomic writes
  - Length and alignment granularities
  - Thin provisioning management
- NVM.FILE Mode
  - Targeted for file based apps.
  - Discovery and use of atomic write features
  - Discovery of granularities



Programming Models Supported in PMDK (2)

- NVM.PM.VOLUME Mode
  - Software abstraction for persistent memory hardware
  - Address ranges
  - Thin provisioning management
- NVM.PM.FILE Mode
  - Application behavior for accessing PM
  - Mapping PM files to application address space
  - Syncing PM files



Programming Models Supported in PMDK (3)

- Uses for NVM.PM.VOLUME
  - Kernel modules
  - PM aware file systems
  - Storage stack components
- Uses for NVM.PM.File
  - Applications
    - Persistent datasets, directly addressable, no DRAM footprint
    - Persistent caches (warm cache effect)
  - Reconnect-able BLOBs of persistence
    (Binary Large Object – set of binary data stored as a single entity)
    - Naming
    - Permissions

Current Status of Support PM Programming Model by Linux

- On Linux, ext4/5 and XFS that support DAX, DO NOT support user space flushing safe w/ CLWB + SFENCE executed by user-space code.
- As an interim solution, Linux provides Device-DAX which allows user space application to open a persistent memory device (w/o file system), memory map it, and utilize user-space flushes to make stores persistent.

Current State of Ecosystem

OS Detection of NVDIMMs	**ACPI 6.0+**
OS Exposes pmem to apps	***DAX* provides SNIA Programming Model** Fully supported: • Linux (ext4, XFS) • Windows (NTFS)
OS Supports Optimized Flush	Specified, but evolving (ask when safe) • Linux: **unsafe** except Device DAX • (and new file systems like **NOVA**) • Windows: **safe**
Remote Flush	Proposals under discussion (works today with extra round trip)
Deep Flush	Upcoming Specification
Transactions,Allocators	Built on above via libraries and languages: • http://pmem.io **Much more language support to do**
Virtualization	All VMMs planning to support PM in guest (KVM changes upstream, Xen coming, others too…)

SNIA (Storage Networking Industry Association) Standards

- SNIA NVM Programming Model
  - Describes the basic programming model used for accessing persistent memory, implemented in PMDK.
- ACPI Spec
  - Starting with version 6.0, defines the *NVDIMM Firmware Interface Table* (NFIT) which is how the existence of persistent memory is communicated to operating systems. The specification also describes how NVDIMMs are partitioned into *namespaces*, methods for communicating with NVDIMMs, etc.
- UEFI Spec
  - Covers other NVDIMM-related topics such as the *Block Translation Table* (BTT) which allows an NVDIMM to provide block device semantics.
- NVDIMM Namespace Spec
  - Described the namespace and BTT mechanisms.
- NVDIMM Driver Writers Guide
  - Adhere to the NFIT tables in the ACPI V6.0 specification, the Device Specific Method (DSM) specification and the NVDIMM Namespace Specification.
- NVDIMM DSM Interface
  - Targeted to writers of BIOS and OS drivers for NVDIMMs whose design adheres to the NFIT Tables in the ACPI specification. The document specifically discusses the NVDIMM Device Specific Method (_DSM) example.

Use Cases Supported by Persistent Memory

- In-Memory Database: Journaling, reduced recovery time, Ex-large tables
- Traditional Database: Log acceleration by write combining and caching
- Enterprice Storage: Tiering, caching, write buffering and meta data storage
- Virtualization: High VM consolidation with great memory density
- High-Performance Computing: Check point acceleration and/or elimination

Questions

- How to partition NVDIMM to support PMEM and BLK per concrete use cases, i.e. how much capacities for PMEM, and how much capacities for BLK?

Reference

- https://software.intel.com/en-us/persistent-memory
- https://nvdimm.wiki.kernel.org/

Backup

Risk in “Happens-before” and “Sync-with”

- Due to compiler self optimization, so the assignment statements not happened as source view.
- However there exist an atomicity “flag” be used, maybe data assigned by Thread A is still resided in cache, not updated in Thread B core’s cache.
- Under multiple core env, using atomicity to protect non-atomical sync is NOT reliable.

```
int a, b; void foo() {
a = 42;
b = a;
assert(b == 42); }
```

```
mov 42, %eax
/* The following 2 statements may execute simultaneously */
mov %eax, (b)
mov %eax, (a)
```

```
int data; std::atomic_bool flag { false };
// Execute in thread A
void producer() {
data = 42; // (1)
flag.store( true);  // (2)
}
// Execute in thread B
void consume()
{ while (!flag.load()); // (3)
assert(data == 42); // (4)
 }
```

6 Memory Order Model in C++ 11

Memory Order 值	**Memory Order 类型**
memory_order_relaxed	Relaxed
memory_order_consume	Consume
memory_order_acquire	Acquire
memory_order_release	Release
memory_order_acq_rel	Acquire/Release
memory_order_seq_cst	Sequentially consistent