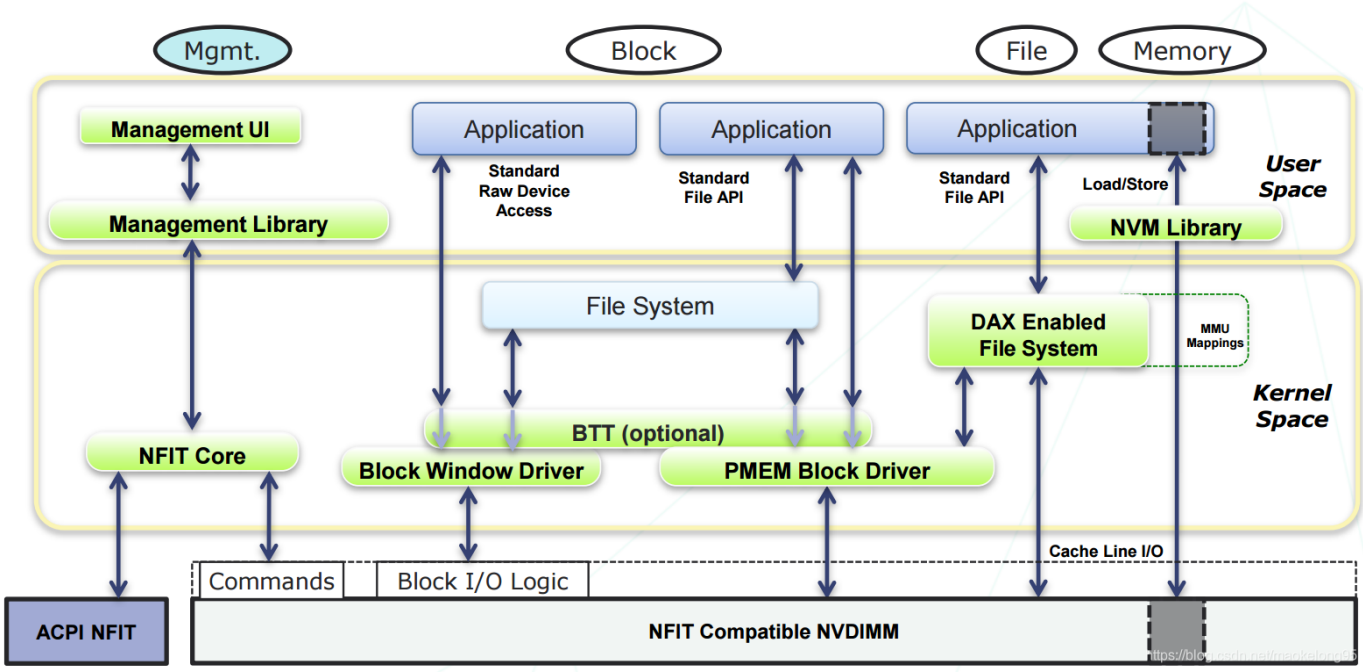
Linux Memory Management：The Function and the Implementation of DAX（Direct Access）Mechanism

1. DAX 简述

直接访问（Direct Access，DAX） 机制是一种支持用户态软件直接访问存储于**持久内存（Persistent Memory，PM）** 的文件的机制，用户态软件无需先将文件数据拷贝到**页高速缓存（Page Cache）**[1]。

上述描述对应到下面这张图（Typical NVDIMM Software Architecture[2]）中，就是说（File）和（Memory）这两条 IO 路径都能绕过页高速缓存。

- 其中 File 路径（下称普通文件路径）表示，用户态软件通过标准文件接口（Standard File API）访问持久内存文件系统。
- 其中 Memory 路径（下称映射文件路径）表示，用户态软件通过映射文件（Memory-mapped File）直接访问 PM。



2. DAX 的原理

以下将结合 Linux v5.8-rc1 中的 XFS 为例进行介绍。

2.1. 普通文件路径如何旁路页缓存

以文件写路径为例，其由 xfs_file_write_iter 提供，该函数部分代码如下所示：

```
STATIC ssize_t
xfs_file_write_iter(
        struct kiocb            *iocb,
        struct iov_iter         *from)
{
        struct file             *file = iocb->ki_filp;
        struct address_space    *mapping = file->f_mapping;
        struct inode            *inode = mapping->host;
        ssize_t                 ret;

        if (IS_DAX(inode))
                return xfs_file_dax_write(iocb, from);

        if (iocb->ki_flags & IOCB_DIRECT) {
                ret = xfs_file_dio_aio_write(iocb, from);
                if (ret != -EREMCHG)
                        return ret;
        }

        return xfs_file_buffered_aio_write(iocb, from);
}
```

这段代码在执行写操作的时候，将分别处理三种情况：

- **DAX**
  - 在将文件系统挂载到 PM 设备时，若设置 DAX 标识（`mount -o dax`），则持久内存文件系统将为所有写操作采用该路径；
  - 该路径主要调用 dax_iomap_rw。该函数在通过 dax_direct_access 获取目标物理内存的地址后，通过 dax_copy_from_iter 调用 NVDIMM 驱动直接把数据拷贝到目标物理内存中，并冲洗（Flush）相应缓存行（Cache Line）。
- DIO（Direct IO）：
  - 在打开文件时，若设置直接 IO 标识（`O_DIRECT`），则文件操作将采用该路径；
  - 该路径主要调用 iomap_dio_rw。该函数仍通过传统存储栈（Storage Stack）访问设备，即通过构造 bio，将请求传递到块设备层（Block Device Layer），再由块设备层调用驱动从而访问设备。
- 正常 IO：
  - 常规的、使用页缓存的 IO 方式。
  - 该路径主要调用 iomap_file_buffered_write。该函数首先通过 pagecache_get_page 获取页缓存（有关页缓存机制的最新设计，可阅读[345]），接着通过 iomap_read_page_sync 封装 bio，以请求块设备层调用驱动，将设备上的数据读取到页缓存中。在准备好页缓存之后，调用

iov_iter_copy_from_user_atomic 将用户态软件请求写入的数据拷贝到页缓存中。一切完成之后，通过 iomap_set_page_dirty 将页缓存设置为脏页。如此迭代，直至用户所有数据都写入页缓存，最后通过 balance_dirty_pages_ratelimited 酌情使用后台进程将脏页回写到块设备中。

2.2. 映射文件路径如何旁路页缓存

调用 mmap 时，文件系统仅仅在进程的 mm_struct 中注册了一段使用虚拟内存区域（Virtual Memory Area，VMA）描述的虚拟地址。后续当用户态软件首次访问映射文件时，内存管理单元（Memory Managment Unit）发现页表项（Page Table Entry,PTE）为空，于是触发 14 号故障，即页故障（Page Fault），使得操作系统开始执行请求调页（Demand Paging）。此时，由虚拟内存管理器（Virtual Memory Manager）和文件系统共同管理页表，以建立虚拟内存到物理内存之间的映射关系。注意，以上为同步过程，而非异步过程，因为页故障是一个异常（Exception）而非软/硬件中断（Software/Hardware Interrupt）。

2.2.1 调用 mmap 时发生了什么

在执行 mmap 系统调用时，主要执行 do_mmap 中的 mmap_region，其根据用户态软件的请求，返回一个用于描述一段可用进程虚拟地址空间的 VMA，接着通过 call_mmap 执行文件系统注册的 mmap 实现，最后将该段 VMA 之添加在进程的 mm_struct 中。

XFS 中 mmap 由 xfs_file_mmap 实现，其中主要语句就一个：`vma->vm_ops = &xfs_file_vm_ops;`。它告诉异常处理例程（Exception Handler）应该调用 `xfs_file_vm_ops` 中相应的函数处理页错误。

2.2.2 请求调页时发生了什么

请求调页主要执行 __xfs_filemap_fault，其代码如下所示：

```
static vm_fault_t
__xfs_filemap_fault(
        struct vm_fault         *vmf,
        enum page_entry_size    pe_size,
        bool                    write_fault)
{
        struct inode            *inode = file_inode(vmf->vma->vm_file);
        struct xfs_inode        *ip = XFS_I(inode);
        vm_fault_t              ret;

        trace_xfs_filemap_fault(ip, pe_size, write_fault);

        if (write_fault) {
                sb_start_pagefault(inode->i_sb);
                file_update_time(vmf->vma->vm_file);
        }
```

```
        xfs_ilock(XFS_I(inode), XFS_MMAPLOCK_SHARED);
        if (IS_DAX(inode)) {
                pfn_t pfn;

                ret = dax_iomap_fault(vmf, pe_size, &pfn, NULL,
                                (write_fault && !vmf->cow_page) ?
                                 &xfs_direct_write_iomap_ops :
                                 &xfs_read_iomap_ops);
                if (ret & VM_FAULT_NEEDDSYNC)
                        ret = dax_finish_sync_fault(vmf, pe_size, pfn);
        } else {
                if (write_fault)
                        ret = iomap_page_mkwrite(vmf,
                                        &xfs_buffered_write_iomap_ops);
                else
                        ret = filemap_fault(vmf);
        }
        xfs_iunlock(XFS_I(inode), XFS_MMAPLOCK_SHARED);

        if (write_fault)
                sb_end_pagefault(inode->i_sb);
        return ret;
}
```

显然，这段代码有两条分支：

- **DAX**：
  该路径主要调用 dax_iomap_fault。该函数首先通过 grab_mapping_entry 获取页缓存中的 DAX Exception Entry（详见[3]），接着通过 xfs_bmbt_to_iomap 准备一个名为 struct iomap 的数据结构。

```
struct iomap {
        u64                     addr; /* disk offset of mapping, bytes
*/
        loff_t                  offset; /* file offset of mapping,
bytes */
        u64                     length; /* length of mapping, bytes */
        u16                     type;   /* type of mapping */
        u16                     flags;  /* flags for mapping */
        struct block_device     *bdev;  /* block device for I/O */
        struct dax_device       *dax_dev; /* dax_dev for dax operations
*/
        void                    *inline_data;
```

```
        void                    *private; /* filesystem private */
        const struct iomap_page_ops *page_ops;
};
```

准备好 struct iomap 之后，通过 dax_iomap_pfn ，结合 struct iomap 所提供的信息，获取目标 PM 页的物理页号（Physical Page Number，pfn）。之后由 dax_insert_entry 将与该页相关联的 DAX Exception Entry 添加到用于维护页缓存的数据结构 XArray 中。最后调用 vmf_insert_mixed_mkwrite 将从 DAX Exception Entry 中缓存的 pfn 填写到对应虚拟页的 PTE 中。

- 正常请求调页
  该路径主要调用 filemap_fault。该函数首先通过 do_sync_mmap_readahead 试图同步地预读文件数据（预读行为可受 madvise 系统调用的影响，因此也可能完全不读取），接着通过 pagecache_get_page 分配页缓存，再通过 xfs_vm_readpage 将读取块设备数据的请求发送到块设备层，从而将文件数据读取到页缓存中。在将文件数据拷贝到页缓存之后，取决于映射文件的类型（MAP_SHARED、MAP_PRIVATE）执行不同的分支。最后返回的页保存在 vmf->page 中。
  - 当是 MAP_SHARED，主要调用 do_shared_fault 函数，该函数：
    1. 调用 xfs_file_vm_ops 中注册的 xfs_filemap_page_mkwrite，从而调用 iomap_page_create 在 vmf->page 对应的 struct page 的 private 字段中塞进去一个 struct iomap_page。

       ```
       /*
       * Structure allocated for each page when block size <
       PAGE_SIZE to track
       * sub-page uptodate status and I/O completions.
       */
       struct iomap_page {
               atomic_t                read_count;
               atomic_t                write_count;
               spinlock_t              uptodate_lock;
               DECLARE_BITMAP(uptodate, PAGE_SIZE / 512);
       };
       ```

    2. 调用 finish_fault，该函数最终通过 alloc_set_pte 将 vmf->page 映射到虚拟页上，为此需要设置 PTE，并设置 Reverse Mapping[6] 信息以支持空闲页回收。
  - 当是 MAP_PRIVATE，调用 do_cow_fault。该函数首先通过 alloc_page_vma 为 VMA 分配一个页，该页保存在 vmf->cow_page 中。接着通过 copy_user_highpage 将 vmf->page 中的数据拷贝到 vmf->cow_page 中。最后通过 finish_fault 将 vmf->cow_page 映射到虚拟页上。

附录 1：术语表

- **持久内存（Persistent Memory，PM）**：指能通过访存指令（区别于系统调用）访问、可按字节寻址的（区别于块）非易失存储器（Non-volatile Memory，NVM）[7]。其中可按字节寻址（Byte-addressable）表示每个寻址单位对应一个 PM 单元（而非字或块）。
- **映射文件（Memory-mapped File）**：是一段虚内存逐字节对应于一个文件或类文件的资源，使得应用程序处理映射部分如同访问主内存。[10]进程地址空间由映射文件和匿名内存（Anonymous Memory）组成。

附录 2：DAX 历史沿革

- 2015，Carsten Otte 在 Linux v2.6 中引入 XIP（Execute-in-place）机制[11]。
  XIP 原本用于嵌入式系统，它摒弃了存储栈中的通用块层及驱动层，并旁路了页高速缓存，使得进程可以直接访问只读存储器或基于 Flash 的内存。
- 2014 年，Subramanya R Dulloor 等人在 PMFS 中基于 XIP 机制管理 PM[12]。
- 同年，Matthew Wilcox 改进了 XIP 并提出了名为 DAX 的子系统。
  他在尝试将 XIP 集成到 Ext4 文件系统时，发现 XIP 无法很好地应对竞争条件（Race Conditions）[13]。当多个线程需要同时访问共享资源，且结果依赖于它们执行的相对速度时，便出现了竞争条件[14]。他所做出的最主要的变动，就是使用文件系统的 `get_block` 路径替代 `struct address_space_operations` 中的 `get_xip_mem`操作[15]。

---

1. Corbet J. The future of DAX. https://lwn.net/Articles/717953/, 2017 ↩
2. intel. NVDIMM Namespace Specification. http://pmem.io/documents/NVDIMM_Namespace_Spec.pdf, 2015 ↩ ↩
3. Zwisler R. A multi-order radix tree. https://lwn.net/Articles/688130/, 2016 ↩ ↩
4. Corbet J. The XArray data structure. https://lwn.net/Articles/745073/, 2018 ↩
5. Corbet J. The future of the page cache. http://tinylab.org/lwn-712467/, 2017 ↩
6. McCracken D. Object-based reverse mapping. in: Proceedings of the Ottawa Linux Symposium (OLS'04). Ottawa, Ontario, Canada: July 21–24, 2004. 357~360 ↩
7. Nalli S, Haria S, Hill M D, et al. An analysis of persistent memory use with WHISPER. in: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17). Xi'an, Shanxi, China: ACM, April 8-12, 2017. 135~148 ↩
8. SNIA. NVM Programming Model (NPM) v1.2. https://www.snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf, 2017 ↩
9. https://software.intel.com/content/www/us/en/develop/articles/persistent-memory-faq.html ↩

10. https://zh.wikipedia.org/zh-hans/%E5%86%85%E5%AD%98%E6%98%A0%E5%B0%84%E6%96%87%E4%BB%B6 ↩

11. Corbet J. Execute-in-place. https://lwn.net/Articles/135472/, 2005 ↩

12. Dulloor S R, Kumar S, Keshavamurthy A, et al. System software for persistent memory. in: Proceedings of the Ninth European Conference on Computer Systems (EuroSys'14). Amsterdam, North Holland, Netherlands: ACM, April 13-16, 2014. 1~15 ↩

13. Corbet J. Supporting filesystems in persistent memory. https://lwn.net/Articles/610174/, 2014 ↩

14. 孙钟秀, 费翔林, 骆斌. 操作系统教程. 第4版. 北京市: 高等教育出版社, 2008. 1~509 ↩

15. Wilcox M. Add support for NV-DIMMs to ext4. https://lwn.net/Articles/613384/, 2014 ↩