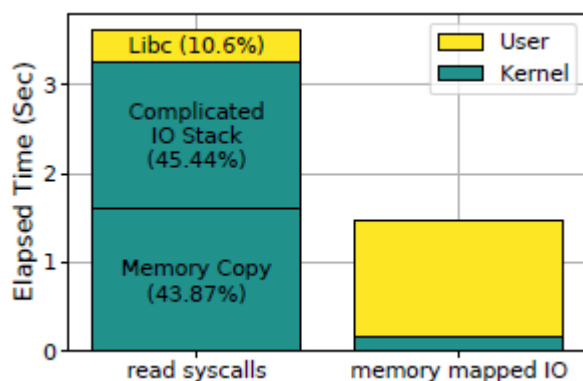


1. 背景

传统的文件系统限制了NVM的性能（software overhead）



2. 目的

充分发挥NVM的高性能

3. 存在的问题

1. 基于kernel的数据访问具有较高的延迟，**mmap io**提供直接NVM访问，能够有效减低kernel开销。

减少了用户空间和page cache之间的数据交换

2. mmap io不提供写数据的原子性，并且为了保证crash-safe，cache line应刷新以确保持久性，并应使用内存隔离以为NVM更新提供正确的持久顺序，这往往会带来大量开销，并且难以编程。
3. 现有的一致性保证机制中CoW存在写放大以及TLB-shootdown问题，journaling (logging) 的两种方式有不同的适应场景。
 - redo log
先将数据写入redo log，再将log持久化到目标文件。redo log中记录最新的数据。（适合写）
 - undo log
先复制目标文件中的数据到undo log中，再对目标文件进行就地更新。目标文件中记录最新的数据。（适合读）

对于可按字节读写的NVM设备，混合日志可显著减少写放大。

4. libnvmio

4.1. 设计目标和实现策略

- 低延时：避免使用内核IO路径。
- 原子性：使用日志维护数据操作原子性
- 高吞吐、高并发：灵活的数据结构、varying sizes and fine-grained logging。

- 以数据为中心，per-block的组织方式：基于inode的log对于同一文件的并发访问不友好。
- 对底层文件系统透明

4.2. Overall Architecture

Libnvmio是一个运行在应用程序所在地址空间的文件库，并且依赖于底层的文件系统。Libnvmio通过拦截IO请求并将其转换成对应的内存操作从而降低软件开销。需要注意的是，Libnvmio只是对数据请求进行拦截，而对于元数据的操作请求则是直接交由内核处理。

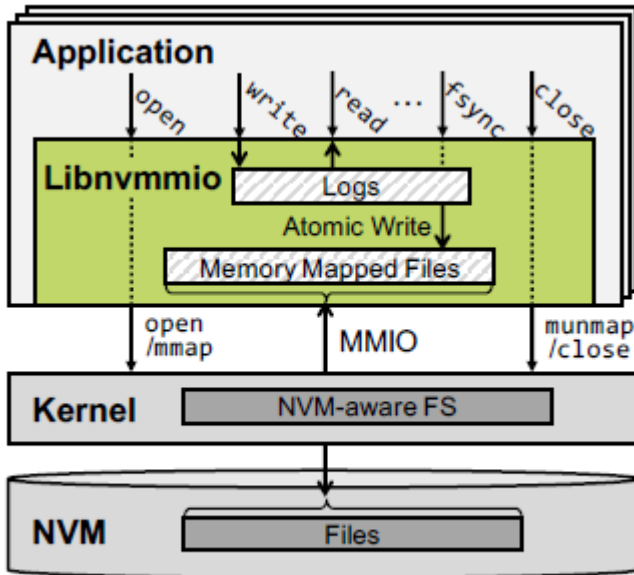


Figure 2: Libnvmio Overview

4.2.1. memory-mapped IO

为了直接访问NVM，libnvmio通过mmap建立文件映射，应分别用memcpy和non-temporal memcpy(MOVNT)来代替read和write方法。有如下两个好处。

- 当持久化和读取数据时，能够避免复杂的内核IO路径
- read/write操作涉及复杂的索引操作来定位物理块。而通过mmap io在建立映射后通过内存映射地址和偏移即可访问文件数据。而且也并不需要通过MMU和TLB完成到物理空间的映射，减少了大量的CPU开销。

4.2.2. 用户级logging

即通过用户级日志记录来提供原子性。有以下两个优点。

- 粒度更小，即使极少量的数据写入也不会产生写放大。
- 不需要通过对TLB中的脏位来进行判断写回。

4.2.3. 应用透明

即能够很容易的对使用write/read方法的应用程序进行修改。并且对于不需要保证原子性的IO操作提供了POSIX版本的memcpy。支持原子性的函数命名统一添加nv前缀（如nvmmmap，nvmemcpy等）

4.3. Scalable Logging

Libnvmio中的日志是以数据块为单位的（per-thread和per-transcaion的日志不利于线程间的数据共享）。在每次需要对文件数据进行更新时，通过需要更新的数据大小来决定log entry的大小（4KB~2MB），并且对所有线程可见。当其他线程需要读取更新处的数据时，直接读取对应的redo log即可。而per-thread的log机制，则需要统计所有线程的log来统计对同一数据块的更新，大大地提高了共享数据访问的性能。

而对于这种具有不同log size的log机制，Libnvmio通过固定深度的radix tree来对索引进行组织，通过虚拟地址来对log entry进行索引。这种多级索引结构对于大量的log相较于索引表能够减少空间开销。而且固定级数能够有效实现无锁机制，相较于平衡树能够提供更好的并行性。

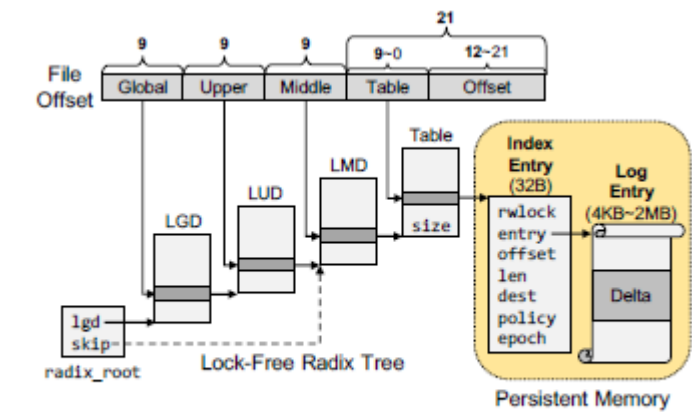


Figure 3: Indexing structure of Libnvmio.

index entry中部分成员解释

- entry: 指向对应的log entry
- offset: updated data在log entry中的起始偏移
- len: log entry中的有效数据
- policy: 使用的log策略 (redo、undo)
- dest: 与offset一同记录的mmap file中对应的地址
- epoch: 用于判断是否已被提交

上图展示了Libnvmio的索引结构。每一个内部节点都是指向下一级内部节点的桶阵列。文件偏移中（实为虚拟地址）的每9位用于在相应的内部节点定位bucket。每一个叶子节点对应一个索引条目（index entry），其有一个指向对应日志条目（log entry）的指针，并且还记录了一些log entry相关的数据，例如更新数据的偏移以及有效长度、读写锁等。

地址中的最后21位由于table以及index entry的索引。根据4KB~2MB的log entry大小，可以很容易地推断出两者的对应关系。如下表（最低位为第0位）。

log size	bits for table	bits for index_entry
4KB	12-20	0-11
8KB	13-20	0-12
16KB	14-20	0-13
...
1MB	20	0-19

log size	bits for table	bits for index_entry
2MB	nul	0-20

4.4. Epoch-based Background Checkpointing

Libnvmmio中的log entries通过显示调用SYNC来进行提交（以文件为单位）。被提交的entries需要被持久化到对应的文件中（称为checkpoint）。为了避免因在关键路径上进行checkpoint而导致性能的降低，Libnvmmi通过创建一个后台线程定期的判断并checkpoint已提交的日志条目。**在后台线程checkpoints时，并不需要获取整颗索引树的读写锁，只需要对相应的log entries进行上锁。**

当SYNC被显示调用时，libnvmmio需要将对应的logs转换成committed的状态。为了减少开销，Libnvmmio基于epoch来进行commit和checkpoint。

Libnvmmio包括两种类型的epoch：

- 由文件的元数据维护的global epoch number
- 由index entry维护的epoch number

每次申请一个新的index entry时，会将其epoch赋值为global epoch。在每次调用SYNC，将global epoch加1，此时并不一定回将对应的log entries写回，需要判断log policy是否改变（后文介绍）。这样，后台同步线程可以通过epoch来判断对应的log entries是否是已被提交的但是未checkpoint的。

```
epoch < global epoch -----> committed
epoch = global epoch -----> uncommitted
```

Per-File Metadata

Libnvmmio在PM中维护了两种元数据:

- index entry(metadata for log entry)
- uma(metadata for Per-File)

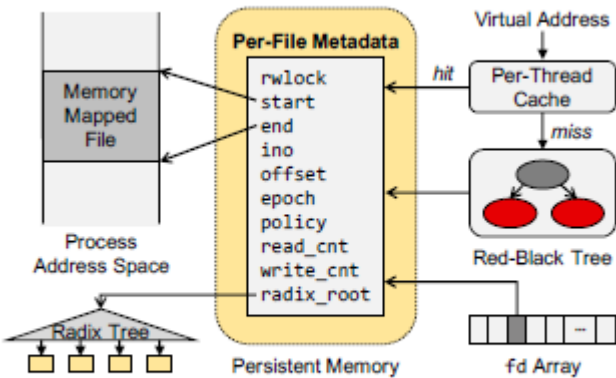


Figure 4: Per-File Metadata

Per-File Metadata中的部分成员解释

```

start: mmap file的起始地址
end: mmap file的终止地址
epoch: global epoch
offset: 映射文件的偏移
read_cnt: 处理的读请求次数
radix_root: 指向全局的radix root

```

当libnvmio访问一个文件时，首先需要获取其元数据。Libnvmio用红黑树来对uma进行组织，在查找时，通过判断虚拟地址是否包含在start-end中来进行查找。为了加快查找策略，Libnvmio申请了一块静态数组来充当cache。查找uma时首先在cache中操作，如果查找不存在，再进入到红黑树中进行查找。在每次在红黑树中查找成功后，都需要将其哈希到静态数组中以加快下一次的查询。同时，Libnvmio还支持通过文件描述符来快速的查找到对应的uma。（Libnvmio维护了一个fd_table[]数组，记录了各种信息，包括对应的uma）

4.5. Hybrid Logging

Libnvmio为了面对不同的读写密集情况，对不同的文件采用log policy（undo or redo）。

- 对于读密集的情况使用undo log
- 对于写密集的情况使用redo log

在每次对文件进行读写时，都需要将元数据中记录的read或者write次数加1。在进行SYNC时，通过判断read/write的值来判断是否需要改变日志策略。如果需要的话，则需要对相应的log entries进行checkpoint，保证此时该文件对应的log entries全部被释放，再修改log policy，这样，在下次申请index entries时，转而使用新的log policy。转换过程如下图。

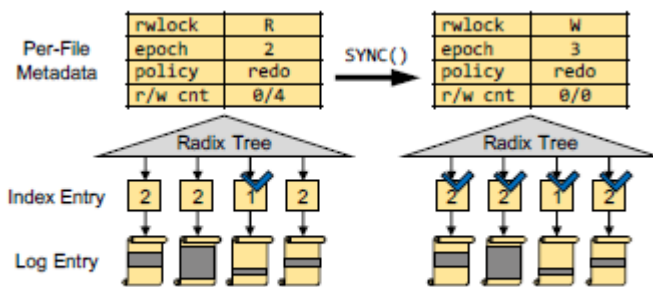


Figure 5: Epoch-based committing

代码分析

数据结构

log_entry_struct: 索引条目结构（index entry），被持久化到PM上，对应文件

`$pmem_path/.libnvmio-$libnvmio_pid/entries.log`

```

typedef struct log_entry_struct {
    union {
        struct {
            unsigned long united;
        };
    };
};

```

```

    struct {
        unsigned long epoch : 20; // 版本号
        unsigned long offset : 21; // 有效数据在log_entry中的偏移
        unsigned long len : 22; // 有效数据的长度
        unsigned long policy : 1;
    };
};
void *data; // 指向log entry
void *dst; // 与offset一起指向写回到映射文件的地址
pthread_rwlock_t *rwlockp;
} log_entry_t;

```

uma_t: Per-File Metadata, 被持久化到PM上, 对应文件

\$pmem_path/.libnvmio-\$libnvmio_pid/uma.log

```

typedef struct mmap_area_struct {
    unsigned long epoch; // 全局版本号
    unsigned long policy; // 日志策略
    void *start; // mmap file起始地址
    void *end; // mmap file终止地址
    unsigned long ino; // inode
    off_t offset; // 一般为0, 代表为文件起始处开始映射
    unsigned long read; // 处理读请求的数据
    unsigned long write;
    struct thread_info_struct *tinfo; // 未使用
    pthread_rwlock_t *rwlockp;
    struct rb_node rb; // 在rbtree中的节点
    struct list_head list; // 同步线程链表中的元素(未使用)
    int id;
    pthread_t sync_thread; // 用于同步的后台线程
} uma_t;

```

fd_mapaddr_struct: 记录文件相关的信息, 以文件描述符为索引, 利用该结构体数组可以通过fd快速查询到文件的相关信息, 例如文件的元数据信息uma。

```

/**
 * @brief 记录文件相关的数据
 *
 */
typedef struct fd_mapaddr_struct {
    void *addr; // 记录映射起始地址
    off_t off; // 文件内偏移
    char pathname[PATH_SIZE];
    size_t mapped_size; // 映射空间的大小, 一般不变, 用于unmap时的参数
    size_t written_file_size; // 映射文件的有效数据长度
    size_t current_file_size; // 文件在nvm上的大小
    int dup; // 记录复制的文件描述符次数
    int dupfd; // 指示当前的fd是否是dup来的。如果fd_table[fd].dupfd != fd.则说明通过调用nvdup产生的fd。

```

```
int open; // 文件被打开的次数，即打开同一文件产生的不同的文件描述符的个数（不包括dup）
int increaseCount; // 文件在nvm上空间扩展的次数，初始值为1
uma_t *fd_uma;
} fd_addr;
```

log_table_struct: 索引树radix tree中的节点结构。

```
/**
 * @brief radix tree的内部节点
 *
 * @param count 具有的子节点树
 * @param type LGD、LUD、LMD前三层。TABLE: 存放index entry的table层
 * @param log_size 指向的log entry的大小
 * @param index 在当前桶阵列的index
 * @param entries 指向的log entries
 *
 */
typedef struct log_table_struct {
    int count;
    log_size_t log_size;
    enum table_type_enum type;
    struct log_table_struct *parent;
    int index;
    void *entries[PTRS_PER_TABLE];
} log_table_t;
```

需要注意的是，radix tree的构建并不是一步到位的，而是每次需要访问到对应的桶阵列（table）时，才从已申请空间的global_table_list中分配。只有当TYPE == TABLE，成员log_size和entries才有意义。

空间分配

相关代码位于[alloctor.c](#)

全局空间链表

```
static freelist_t *global_tables_list = NULL; /* 指向table空间的链表指针 */
static freelist_t *global_entries_list = NULL; /* 指向index entries空间的指针链表指针 */
static freelist_t *global_data_list[NR_LOG_SIZES] = {NULL, }; /* 指向log entries空间的链表指针数组 */
static freelist_t *global_uma_list = NULL; /* 指向uma空间的链表指针 */
```

在每次调用open函数打开一个文件时，都会继续初始化检查。调用init_libnvmio以初始化，函数框架如下：

```
|-- init_libnvmio
    |-- init_env() 设置pmem_path
```

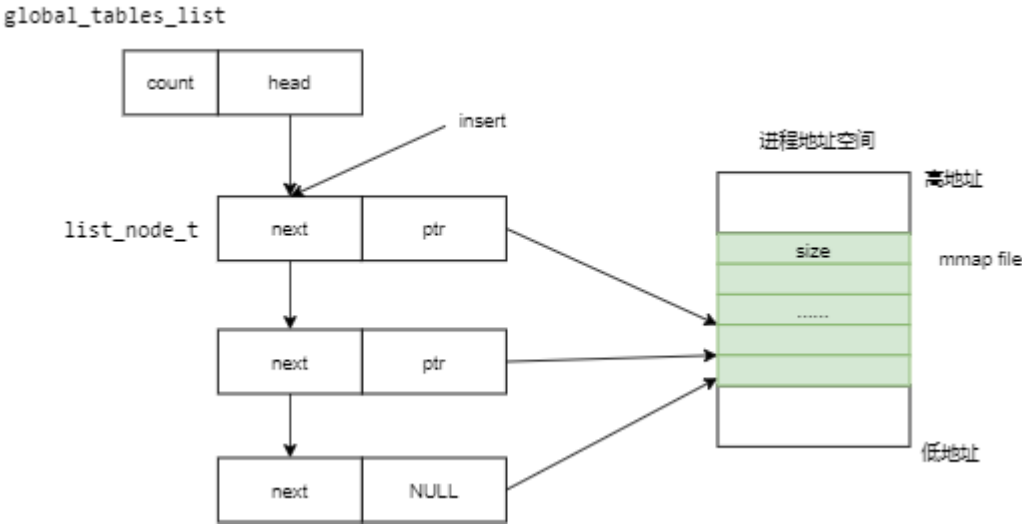
```
|-- init_global_freelist 申请空间
|-- create_global_tables_list 申请
|-- create_global_entries_list
|-- create_global_data_list
|-- create_global_umas_list
|-- init_radixlog 初始化radix tree
|-- init_uma
|-- init_base_address
```

所有global_list的类型都为freelist_struct，结构如下：

```
typedef struct freelist_struct {
    list_node_t *head; // 指向链表头
    unsigned long count; // 链表节点个数
    pthread_mutex_t mutex;
} freelist_t;
```

以create_global_table_list为例，由于table（radix树中的桶阵列）不需要持久化到PM上，于是首先创建一个匿名映射（对于global_entries_list而言，则是先在PM上创建文件并申请相应的空间，然后再建立映射），然后调用create_list创建链表。

在创建链表时，包括链表包含的节点数，之后根据传入的size来确定每一个节点指向的映射空间起始地址，节点从链表头插入。



本地空间链表
空间申请与回收