存储层之间的交互
Storage Layer Interactions

钮鑫涛
南京大学
2024春

本门课由蒋炎岩和钮鑫涛共同构建

“层”之间不是独立的

File descriptor	• sys_file.c
Pathname	• file.c
Directory	• file.c
Inode	• fs.c
Logging	• log.c
Buffer cache	• bio.c
Disk	• ide.c

这些层的实现是相互依赖的，并不是一个个孤立的层

xv6存储层的更精准的视图

⬍ `int file_descriptor, char* filename`

⬍ `struct file`

⬍ `struct inode`

⬍ `struct buf`

⬍ `struct buf`

⬍ `struct buf`

⬍ `in/out device registers`

相邻层之间传递的对象

- 每一层都是仅使用其下层的 API 实现
  - 每一层的公共 API 都在 defs.h 中定义（其他函数是私有的）
- 每一层的 API 必须设计为直接满足其上层的需求

1) 文件描述符层(syscalls，sys_file.c)

```
int read(int, void*, int);
int write(int, void*, int);
int open(char*, int);
int close(int);
int dup(int);
int link(char*, char*);
int unlink(char*);
int fstat(int fd, struct stat*);
int mkdir(char*);
int chdir(char*);
```

- 这是在 user.h 中提供给用户代码的系统调用 API
- 这些API只关联
  - 文件描述符（整数）
  - 字节数组
  - 文件路径字符串

文件描述符

- 一个文件描述符关联一个进程打开的一个文件——主要包括inode、读/写权限和一个offset

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  Inode *ip; // in-RAM copy of inode
  uint off; // file offset
};
```

- 每个进程包括的控制表结构proc struct包含：

```
struct file *ofile[NOFILE];  // Up to 16 open file descriptors
struct inode *cwd;           // Current directory
```

2）虚拟文件系统层 (file.c)

- 就是一些用于处理 `struct file*` 的辅助函数

```
struct file* filealloc(void);
void fileclose(struct file *f);
struct file* filedup(struct file *f);
int fileread(struct file *f, char * data, int size);
int filestat(struct file *f, struct stat *st);
int filewrite(struct file *f, char* data, int size);
```

- 系统调用（sys_file.c层里实现的函数）的实现需要的不仅仅是这些函数
  - 比如，我们仍然没有根据路径字符串打开/创建文件的方法
- 所以这是一个不完整的层

struct inode (内存中)

- 全局 icache 在内核内存中存储50个活跃的inode（在 ref==0 时清理）

```
struct inode {
  uint dev;     // Device number
  uint inum;    // Inode number
  int ref;      // Reference count
  int flags;    // I_BUSY, I_VALID

  short type;
  short major;
  short minor;
  short nlink; // #Hard links
  uint size;
  uint addrs[NDIRECT+1];
};
```

inode在file.h里定义

硬盘上inode信息的拷贝，即dinode结构体

3) inode层 (fs.c)

```
Inode* namei(char* path);          // 根据路径找到inode
Inode* nameiparent(char* path, char* name);  // 得到当前文件所在目录的inode

       // 读/写一个inode所指向的文件
int    readi(Inode*, char* data, uint offset, uint size);
int    writei(Inode*, char* data, uint offset, uint size);

       // 从inode中拷贝状态信息(size, timestamp, etc.) 到stat中
void   stati(Inode*, struct stat*);

       // 将一个新的目录项(name, inum)写到目录dp中
int    dirlink(Inode* dp, char* name, uint inum);

       //从dp中寻找并返回文件名是name的inode，找到的话，poff设定为找到的项的下标
Inode* dirlookup(Inode* dp, char* name, uint* poff);

       //返回ip的第bn个索引指向的块号，如果没有，先分配再返回
int    bmap(struct inode *ip, uint bn);
```

3) inode层 (fs.c)

```
Inode* ialloc(uint, short); //在磁盘里申请一个新的inode，并返回in-memory的拷贝
Inode* idup(Inode*); // 仅仅增加inode的ref的数量
void   ilock(Inode*); // 对inode上锁（互斥），如果有必要（valid为0，），需要从磁盘读取这个inode
void   iunlock(Inode*); //释放inode的锁
void   iput(Inode*); // ref的数量减一，如果为0，inode缓存可以清除，如果inode的nlink为0，那么磁盘清空该inode
void   iupdate(Inode*); // 将内存中改变后的inode更新到磁盘上
```

- 这些返回inode的函数内部都会调用iget(int dev, int inum)，从缓存icache中找到一个相应的inode，这会增加缓存inode的引用ref计数。如果缓存中没有的话，会返回一个初始化的inode（valid为0），后续(调用ilock时)需要从磁盘读取具体的data
- 缓存的 inode 是共享的，因此在访问inode时调用 ilock 和 iunlock。
- 如果修改了 inode，需要调用iupdate来更新磁盘同步信息

struct buf

在buf.h中

```
//I0 Buffer
struct buf {
    int flags;
    uint dev;
    uint sector;
    struct buf*prew; // LRU cache list
    struct buf*next;
    struct buf *qnext; // disk queue
    uchar data[512]:
};
#define B_BUSY 0x1  // buffer is locked by some process
#define B_VALID 0x2 // buffer has been read from disk
#define B_DIRTY 0x4 // buffer needs to be written to disk
```

Busy/Locked? Valid? Dirty?

Doubly-linked circular list of buffers

List of buffers waiting to be written to disk

4) 日志层 (log.c)

- xv6中日志的数据结构

```
struct logheader {
    int n;                          // the num of logs (mark the length of array)
    int block[LOGSIZE];             // the corresponding subscript of buf in the buf array
};

struct log {
    struct spinlock lock;
    int start;                      // the log start at this block
    int size;                       // the max log num
    int outstanding;                // how many FS sys calls are executing.
    int committing;                 // in commit(), please wait.
    int dev;                        // the dev num
    struct logheader lh;
};
```

4) 日志层 (log.c)

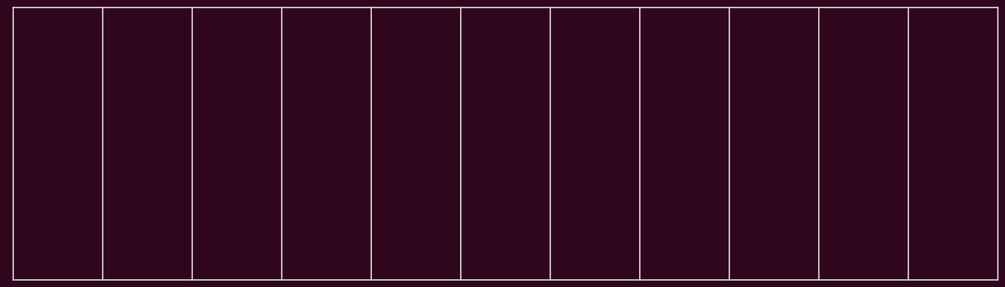
struct log

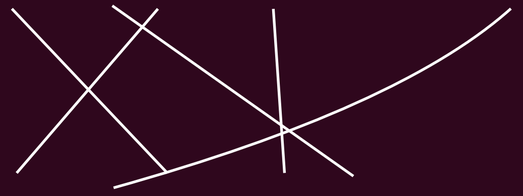
spinlock	lock
start	30
size	size
outstanding	0
committing	1
dev	0
log header	lh

struct logheader

n	5
block	

struct buf buf[NBUF]





3	7	1	4	2	…

int block[LOGSIZE]

4) 日志层 (log.c)

- 日志(Logging)是可选项，一般只会用在文件系统中比较关键的数据区域

```
void begin_op();
```

  - 标志日志开始，如果此时有正在提交的日志或者空间不足，等待

```
void log_write(struct buf*);
```

  - 标记要写入的缓冲块，保留需要写入的块的位置，但暂时不写入磁盘
  - 允许多个写操作合并为一个（写统一缓冲块）

```
void end_op();
```

  - 将日志提交写入磁盘中，接着写入目标扇区.

5) 缓冲区缓存（Buffer cache）层 (bio.c)

```
struct buf* bread(int device, int sector);
```

- 返回磁盘设备device扇区sector的缓冲区缓存（如果已经在缓冲区(bget函数)直接返回，否则才读磁盘）
- 返回的缓冲区会被锁定，供调用的线程专用

```
void bwrite(struct buf* b);
```

- 将写缓冲区的新内容写入磁盘
  - xv6 使用“直写”（write through）策略——始终立即写入
- 在调用 bwrite 前必须先调用bread

```
void brelse(struct buf* b);
```

- 释放该缓冲区的锁

6) 设备驱动层 (ide.c)

```
void iderw(struct buf*);
```

- 如果buf的有效位（valid bit）为0，则从磁盘读取到缓冲区(先在内存申请一个空的buffer，valid 为0)
- 如果buf的脏位（dirty bit）为1，则将缓冲区写入磁盘
- 在返回之前睡眠（on buffer pointer），因此从调用线程的角度来看，它会阻塞调用线程

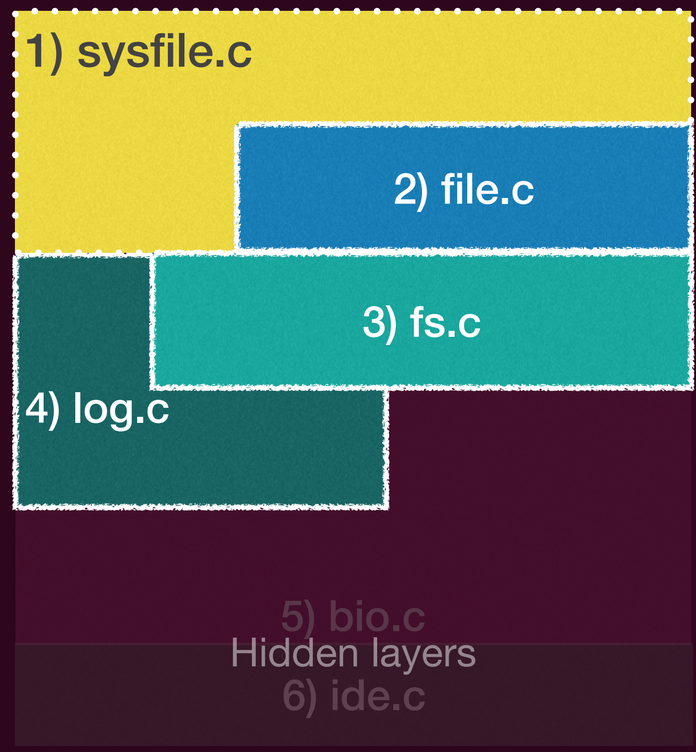
```
void ideintr(void);
```

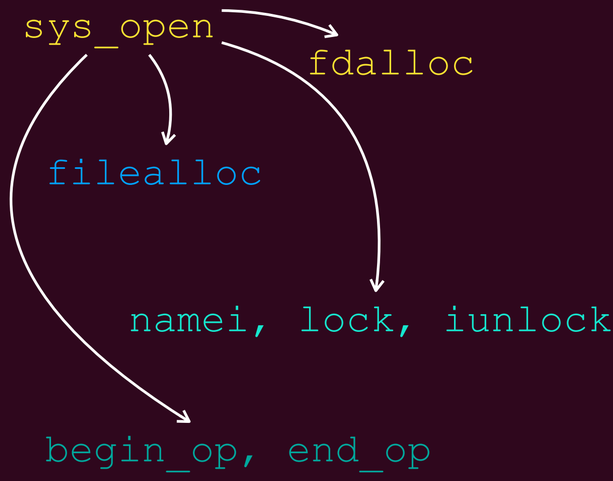
- 磁盘中断处理程序，以获知读/写请求的完成
- 如果需要（读磁盘块请求），读取到缓冲区，并唤醒正在睡眠的线程

整体流程样例

在sysfile.c的open(char*,...)行为

- 假设要打开的文件存在，sys_open会
  - begin_op: 开始日志交易
  - namei: file path → inode
  - ilock: lock inode
  - filealloc: 申请一个空文件结构
  - fdalloc：列出该进程打开的文件并得到文件描述符
  - iunlock: 解锁inode
  - 将inode的数字存储在文件结构里
  - end_op: 关闭交易

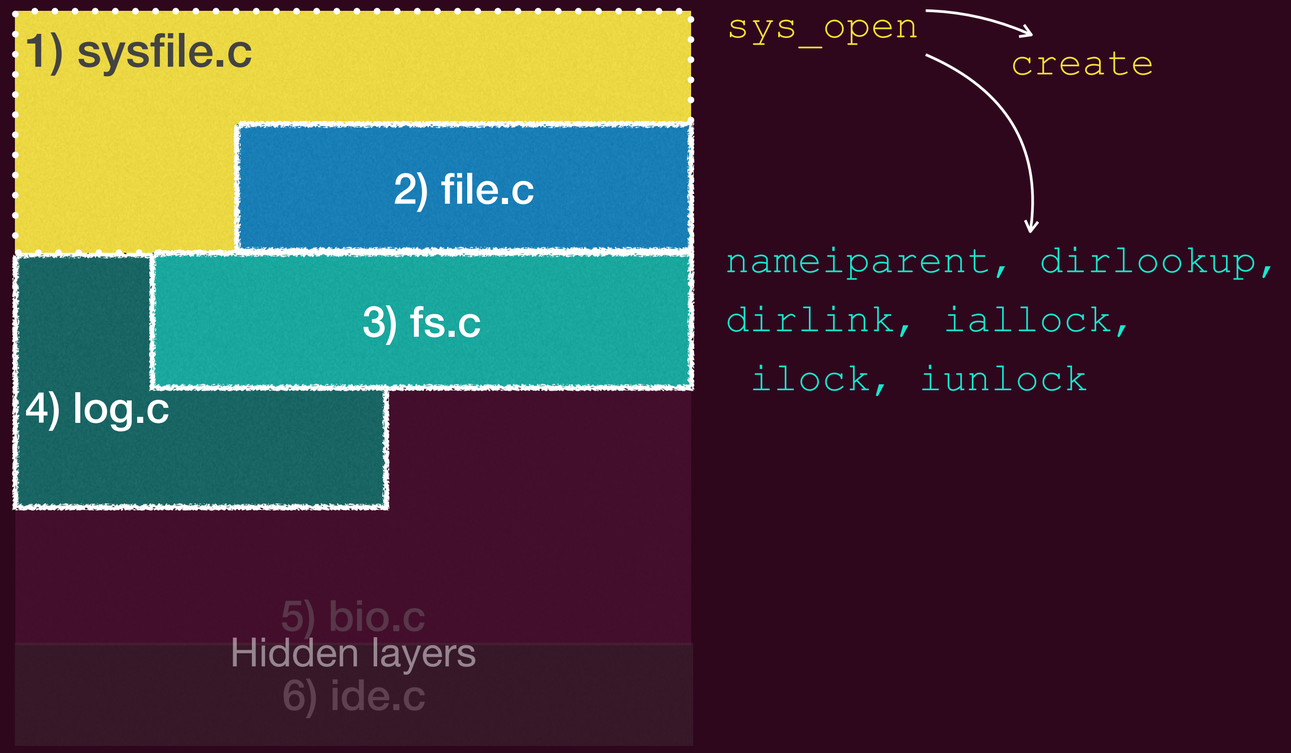




sys_open 的实现只涉及下面三层，更深层次
对于syscall层而言是隐藏的

在sysfile.c的create(char*,...)行为

- 如果用`sys_open`去创建新文件：
  - `nameiparent`: file path → parent inode
  - `ilock`: lock parent inode
  - `dirlookup`: 检查文件是否存在
  - `ialloc`: 为这个新文件申请一个inode
    - `ilock`: lock this inode
    - 设置device number 和 nlink = 1
  - `dirlink`: 将新文件加入到父文件夹的inode中
  - `iupdate`：更新该inode和父inode的信息到磁盘
  - `iunlock`: 释放该inode和父inode的锁

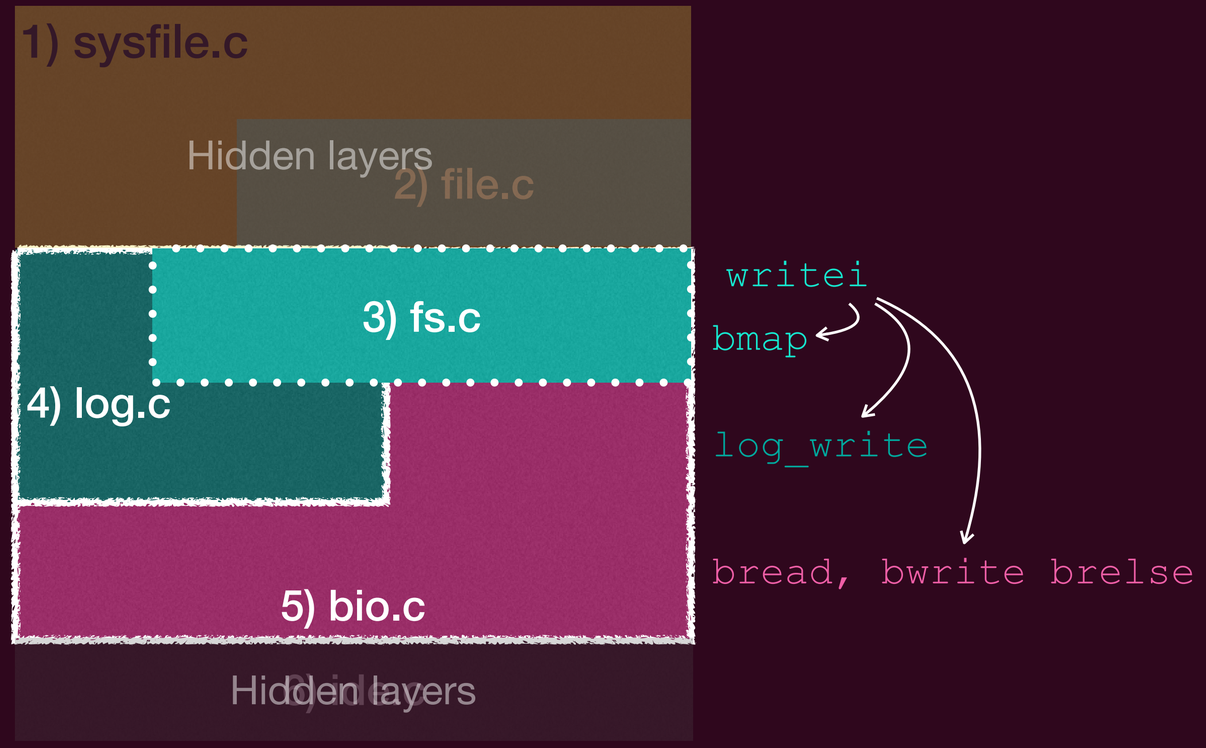


在file.c的filewrite(File*,char *,int n)行为

- sys_write 对struct file的写操作会调用 filewrite，其内部：
  - ilock
  - writei：根据文件对应的inode编号和偏移量来写入对应的字符串
  - iunlock

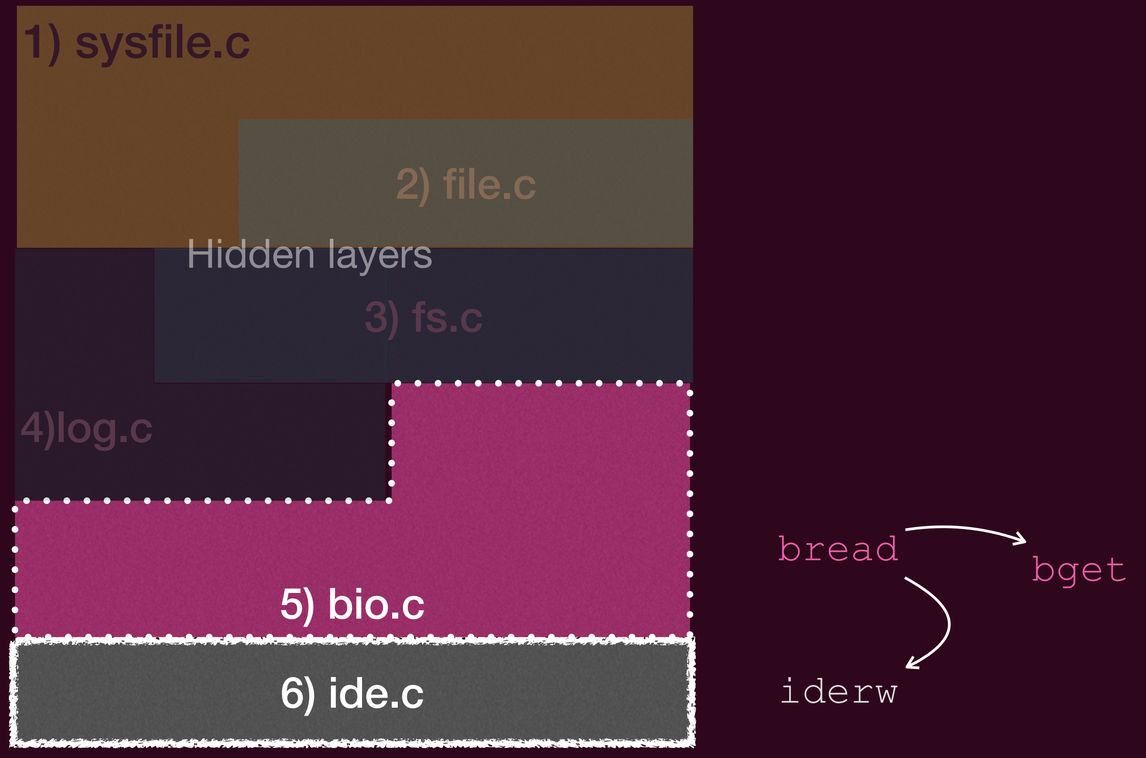
在fs.c中writei(Inode*,char *, int offset, int n)的行为

- 现在我们处于一个较低的层级:
  - `bmap`：返回inode第offset索引的磁盘块号（没有则分配）
  - `bread`: 得到该块号相应的缓冲区（没有的话要从磁盘读），并上锁
  - `log_write`: 将这次的写提交到日志，相应的buf设为脏位
  - `brelse`: 释放缓冲区的锁
  - 最终会被`bwrite`写到磁盘上（日志交易结束）



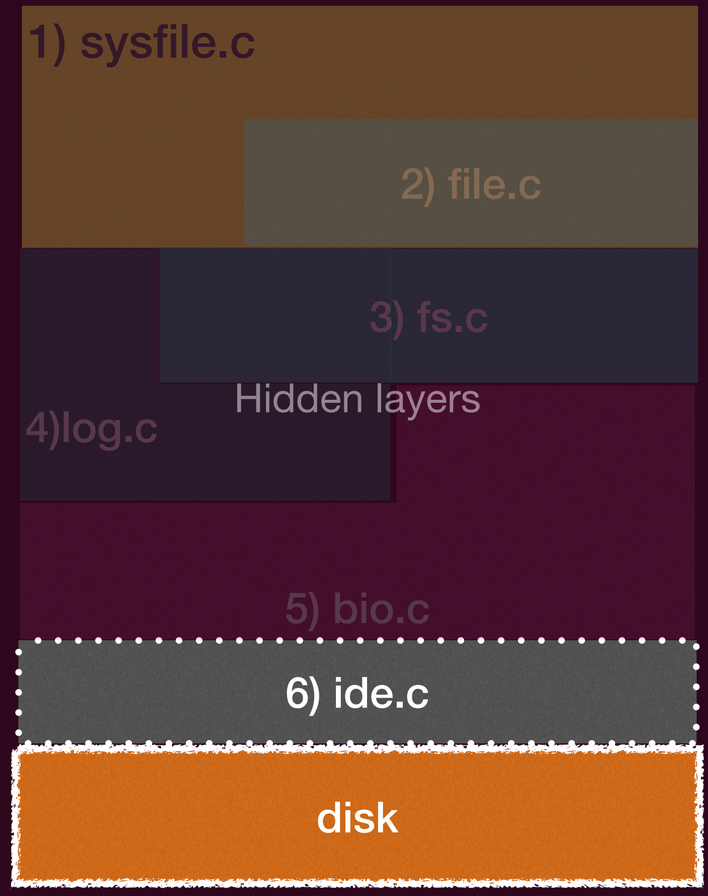
在bio.c中bread(int device, int sector)的行为

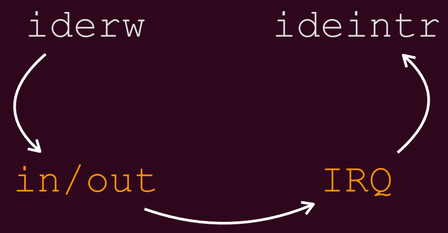
- 当处在buffer层时：
  - `bget(…,int sector)`：得到一个相应扇区的缓存拷贝，或者没有缓存，则返回一个空的buffer（valid = 0）
    - 同时锁住该缓冲区
- 如果是一个空的buffer，那么调用：
  - `iderw`：从磁盘读取相应的扇区



在ide.c中的iderw(struct buf*) 行为

- 最低层，设备驱动层：
  - 使用in/out 汇编指令和相应设备寄存器的端口号
  - 实现一个相应的中断处理函数，ideintr，用来唤醒调用者通知设备处理的状态



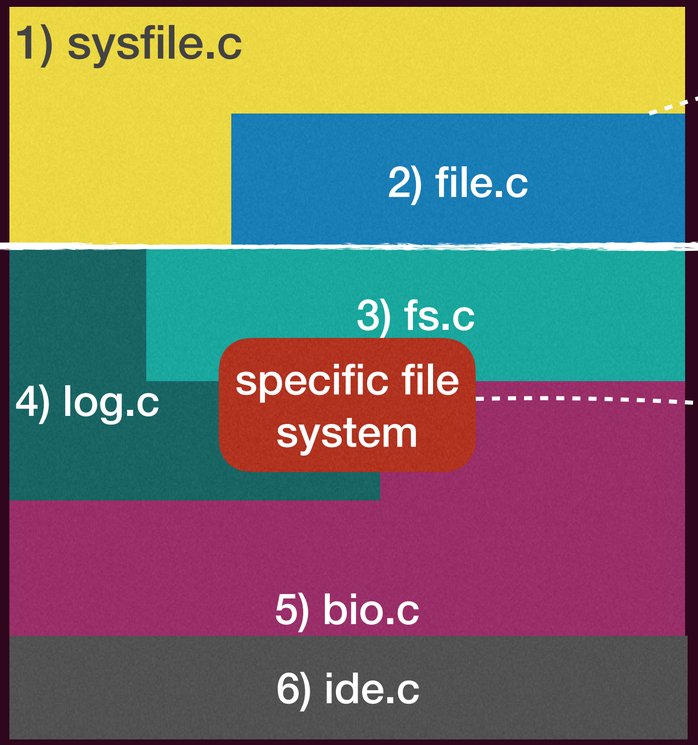


内存映射文件mmap?

- Xv6没有相应的实现，但是作为一个作业
- 有了文件系统和虚拟内存，其实mmap就很自然了
  - 就是建立address 到 file bytes的映射 （可以用某个数据结构存储address的开始和结尾、file的文件描述符和offset）
  - 之后操作文件就变为了操作内存，方便多了
  - 首次访问会出现缺页中断，那么再用文件系统的readi从磁盘读取对应的数据到页面里
- unmap就是解除两者映射，可以根据映射的需求选择是否将更改后的内存写回相应的文件对应的块
- 实际情况需要处理更多复杂情况

支持多个文件系统

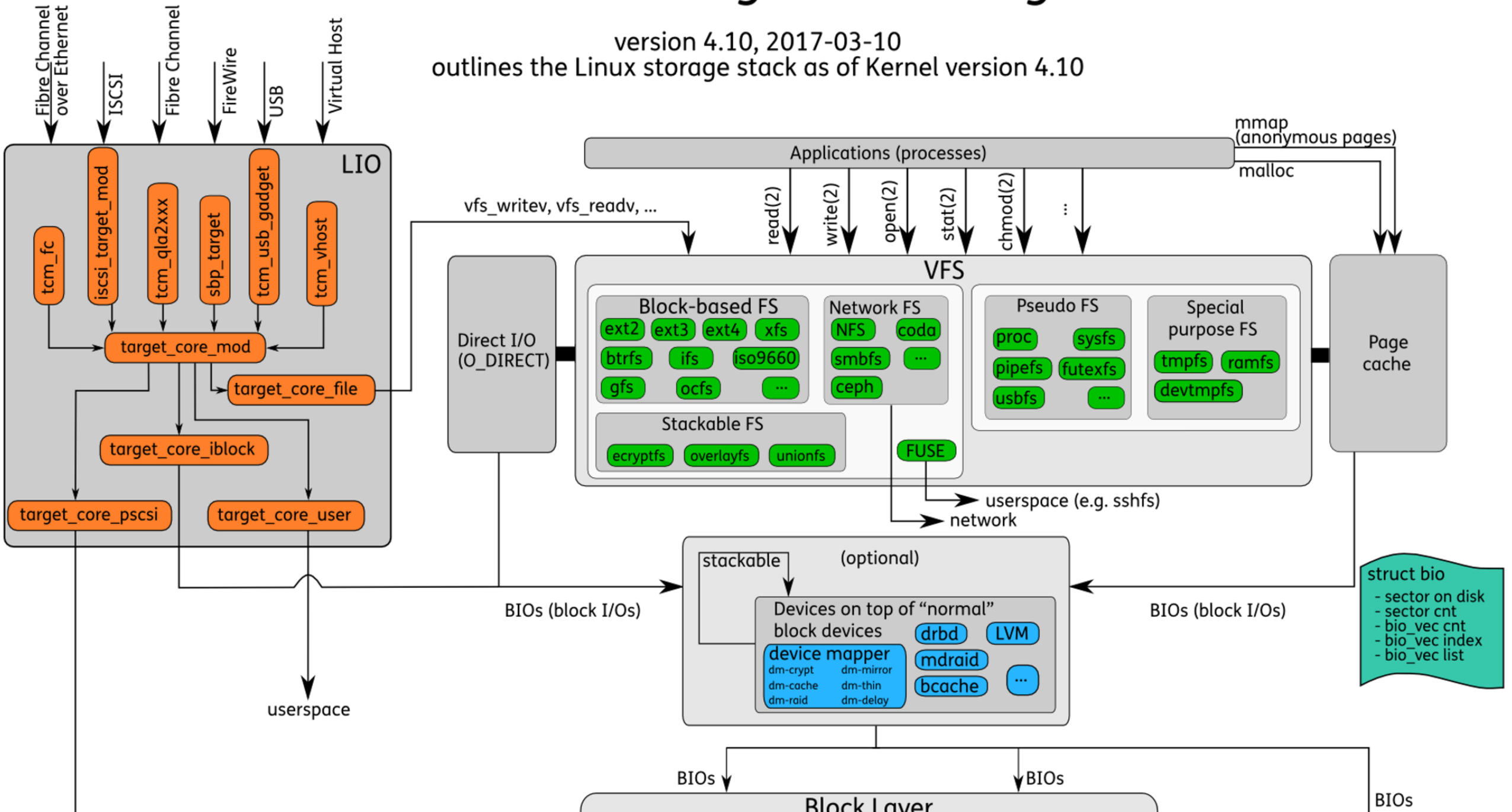
- xv6 只支持一个文件系统
- 系统调用的实现直接与 inode 交互
  - 但有些文件系统甚至不使用 inode!
  - 日志记录的实现也特定于文件系统

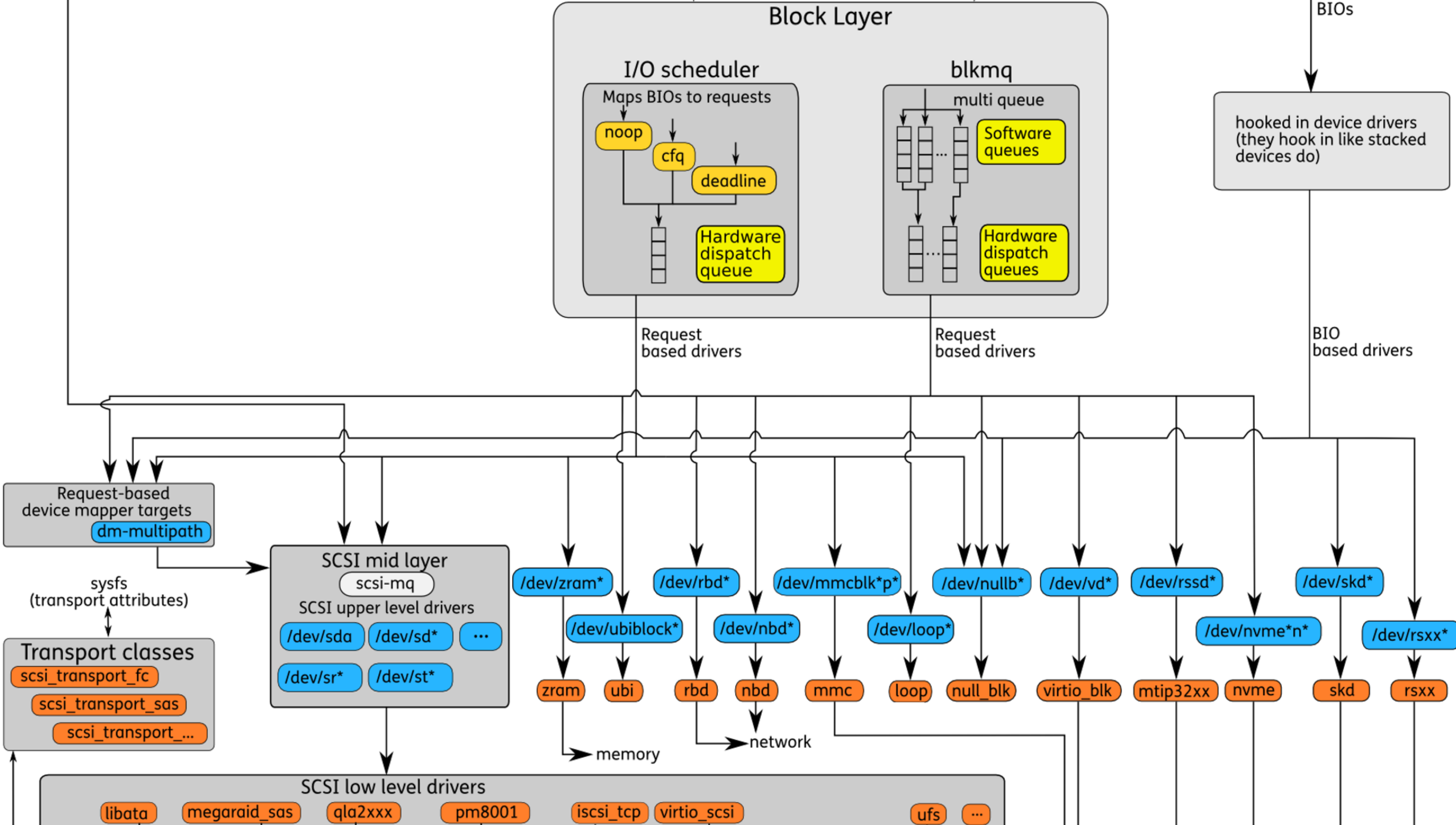


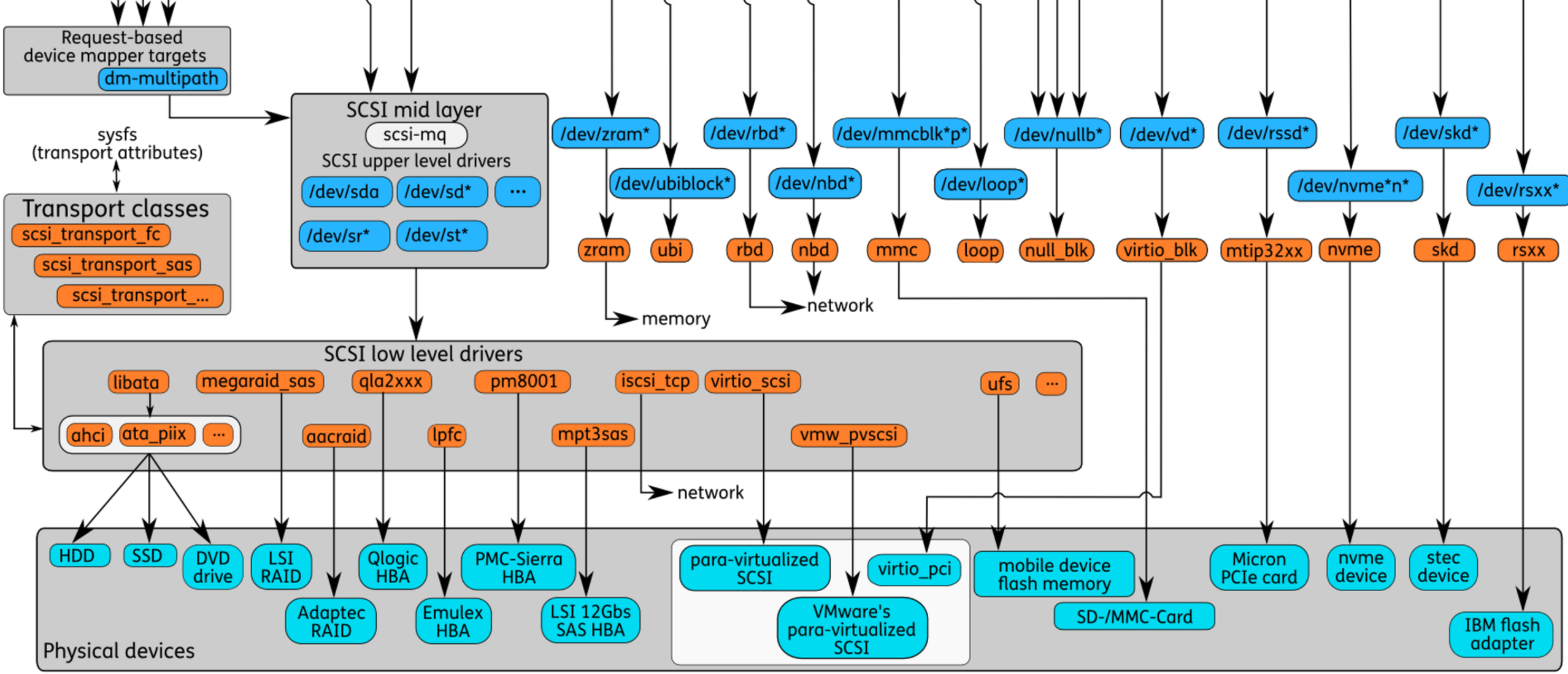
- Linux 有一个虚拟文件系统（VFS）层来支持多个文件系统
  - 上层的系统只调用定义好的VFS的接口，VFS可以将接口绑定具体的文件系统实现
  - 一个文件系统被挂载时，本质上就是向VFS暴露每个接口具体实现函数的入口

The Linux Storage Stack Diagram

version 4.10, 2017-03-10
outlines the Linux storage stack as of Kernel version 4.10




Block Layer
I/O scheduler
Maps BIOs to requests
noop
cfq
deadline
Hardware dispatch queue
blkmq
multi queue
Software queues
Hardware dispatch queues
BIOs
hooked in device drivers (they hook in like stacked devices do)
Request based drivers
Request based drivers
BIO based drivers
Request-based device mapper targets
dm-multipath
sysfs (transport attributes)
Transport classes
scsi_transport_fc
scsi_transport_sas
scsi_transport_...
SCSI mid layer
scsi-mq
SCSI upper level drivers
/dev/sda
/dev/sd*
...
/dev/sr*
/dev/st*
/dev/zram*
/dev/ubiblock*
/dev/rbd*
/dev/nbd*
/dev/mmcblk*p*
/dev/loop*
/dev/nullb*
/dev/vd*
/dev/rssd*
/dev/nvme*n*
/dev/skd*
/dev/rsxx*
zram
ubi
rbd
nbd
mmc
loop
null_blk
virtio_blk
mtip32xx
nvme
skd
rsxx
memory
network
SCSI low level drivers
libata
megaraid_sas
qla2xxx
pm8001
iscsi_tcp
virtio_scsi
ufs
...



THOMAS KRENN®

The Linux Storage Stack Diagram
http://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram
Created by Werner Fischer and Georg Schönberger

总结

- 文件系统各层之间存在依赖
- 每一层的实现基于其下面的层（可能不止一个）
- 下一层的必须为上层提供合适的API
- 更深的层往往对更上面的层隐藏
- Linux 提供了虚拟文件系统 (VFS)来支持一个机器含有多种不同的文件系统

阅读材料

- [OSTEP] 第39, 40, 41, 42, 43, 44章
- xv6代码