EECS-343 Operating Systems
Lecture 17:
Storage Layer Interactions

Steve Tarzia

Spring 2019

Northwestern

Announcements

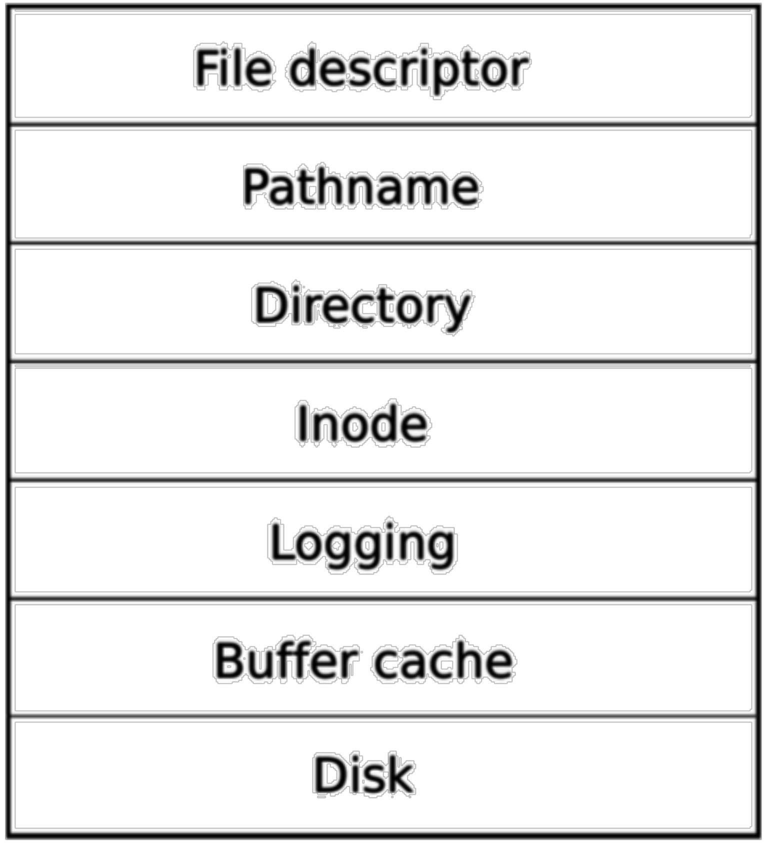
- Project due Wednesday.
- HW due next Friday.
- Exam Monday June 10th at 3pm.

Last Lecture: Buffer Caching & Logging

- Trace of file operations shows that many accesses to disk are needed for even a single open/read/write.
- To improve performance, ***cache*** a small number of active disk blocks
  - Allows later reads to happen in memory
  - Multiple writes can be absorbed and all are immediately visible in memory
- Each buffer is locked by a thread before use
- ***Write-ahead logging*** makes multiple disk writes appear atomic, even if the machine is powered-down in the middle of the transaction.
  - Very important for related changes to inodes & bitmap (metadata in general)
  - Data is written twice: to log first, then to main disk.
  - On reboot, interrupted transaction is either ***rolled back*** or ***replayed***.

xv6 storage: a layered implementation

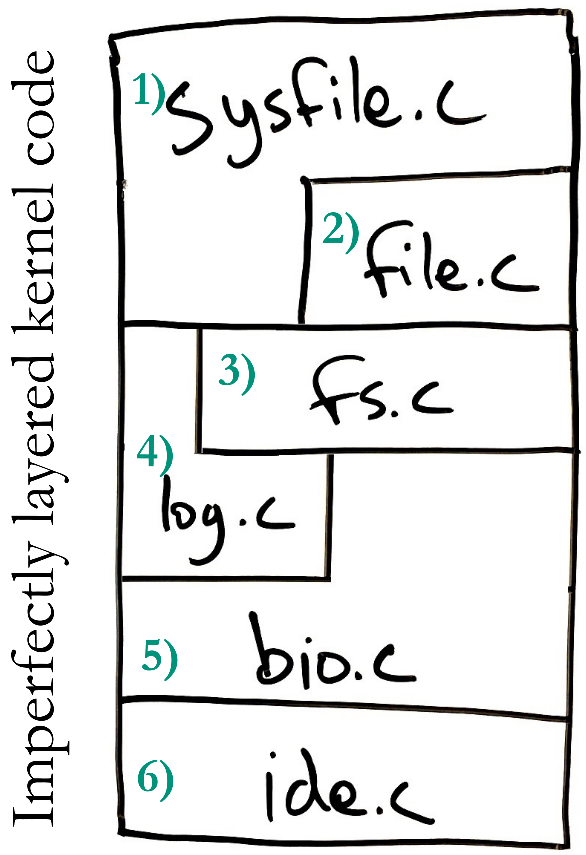
Book's view:



- `sys_file.c`
- `file.c`
- `file.c`
- `fs.c`
- `log.c`
- `bio.c`
- `ide.c`

Unfortunately, the implementations of these layers are *interdependent*. They are not true layers

More accurate view of storage stack in xv6



↕ int file_descriptor, char* filename

↕ struct file

↕ struct inode

↕ struct buf

↕ struct buf

↕ struct buf

↕ in/out device registers

Objects passed between the layers.

- Each layer is implemented using only the layer(s) below's API.
  - Each layer's *public* API is defined in `defs.h` (other functions are *private*).
- Layer's API must be designed to meet needs of the *client* layer(s) directly above.

1) File descriptor level (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*);
```

- This is the system call API presented to user code in user.h.
- Works with
  - integer file descriptors
  - byte arrays
  - file path strings

File descriptors

A **file descriptor** is a file open by a process – an inode, r/w permissions, and a cursor:

```
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/cursor
};
```

Each process' `proc struct` has:

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

2) Virtual file system layer (file.c)

Just a bunch of helper functions for working with **`struct file*`**:

```
File* filealloc(void);
void  fileclose(File*);
File* filedup(File*);
int   fileread(File*, char* data, int size);
int   filestat(File*, Stat*);
int   filewrite(File*, char* data, int size);
```

Syscall implementations needs more than just these functions.

- We still don’t have a way to open/create a file using a path string.
- So this is an incomplete layer.

struct inode (in-memory version)

```
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;                 > copy of disk inode
  uint size;                  |
  uint addrs[NDIRECT+1];      /
};
```

Global *icache* stores 50 active inodes in kernel memory (cleanup when ref==0)

3) Inode layer (fs.c) is where it gets interesting

```
        // get inode corresponding to filename
Inode* namei(char* path);
Inode* nameiparent(char* path, char*);   // get parent inode
        // read/write to a specific location in a file
int    readi(Inode*, char* data, uint offset, uint size);
int    writei(Inode*, char* data, uint offset, uint size);
        // copy file stats (size, timestamp, etc.) from inode
void   stati(Inode*, struct stat*);
        // write a new directory entry (name, inum) to inode
int    dirlink(Inode*, char*, uint);
        // Look for a directory entry in a directory
Inode* dirlookup(Inode*, char* name, uint*);
```

3) Inode layer (fs.c) continued

```
Inode* ialloc(uint, short); // get new inode
Inode* idup(Inode*);  // "copy" inode, but actually just increment ref count
void   ilock(Inode*);  // also read data lazily
void   iunlock(Inode*);
void   iput(Inode*);  // decrement ref count, cleanup
void   iupdate(Inode*);  // tell buffer layer that inode was modified
```

- Functions that return an inode call **`iget(int dev, int inum)`** internally, which increments the cached inode's reference count.
- Cached inodes are *shared*. Call **`ilock`** and **`iunlock`** when accessing. **`iupdate`** if modified. **`iput`** when done.

struct buf

```c
// IO Buffer
struct buf {
  int flags;
  uint dev;
  uint sector;
  struct buf *prev; // 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.
(Implementing it here is kind of sloppy.)

4) Logging (log.c)

Logging is optional, and only used in filesystem critical sections.

```
void begin_op();
```

- Waits until the logging system is not committing and there is enough free space in the log.

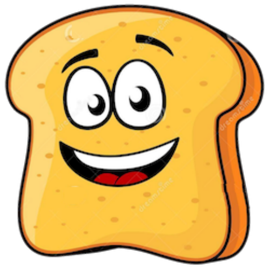
```
void log_write(struct buf*);
```

- Reserves a place for the block in the log, but does not write to disk yet.
- Allows multiple writes to be *absorbed* (combined into one).

```
void end_op();
```

- Writes the transaction to disk in the log, then in the destination sectors.

5) Buffer cache layer (bio.c)



```
Buf* bread(int device, int sector)
```

- Returns a cache buffer with the data at a given location on a given disk.
- Buffer is locked for thread's exclusive use.

```
void bwrite(Buf* b)
```

- Write buffer's new contents to disk.
  - xv6 uses a "write through" cache – we always write immediately.
- Must always call bread before bwrite.

```
void brelse(Buf* b)
```

- Release the lock on the buffer

6) Device driver layer (ide.c)

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

- Read from disk to buffer if valid bit=0
- Write buffer to disk if dirty bit=1
- Sleep (on buffer pointer) before returning, so it's a blocking call from the calling thread's perspective, even though it uses interrupts.

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

- Handles disk interrupts to complete a r/w request.
- Read to buffer (if necessary) and wake sleeping thread.

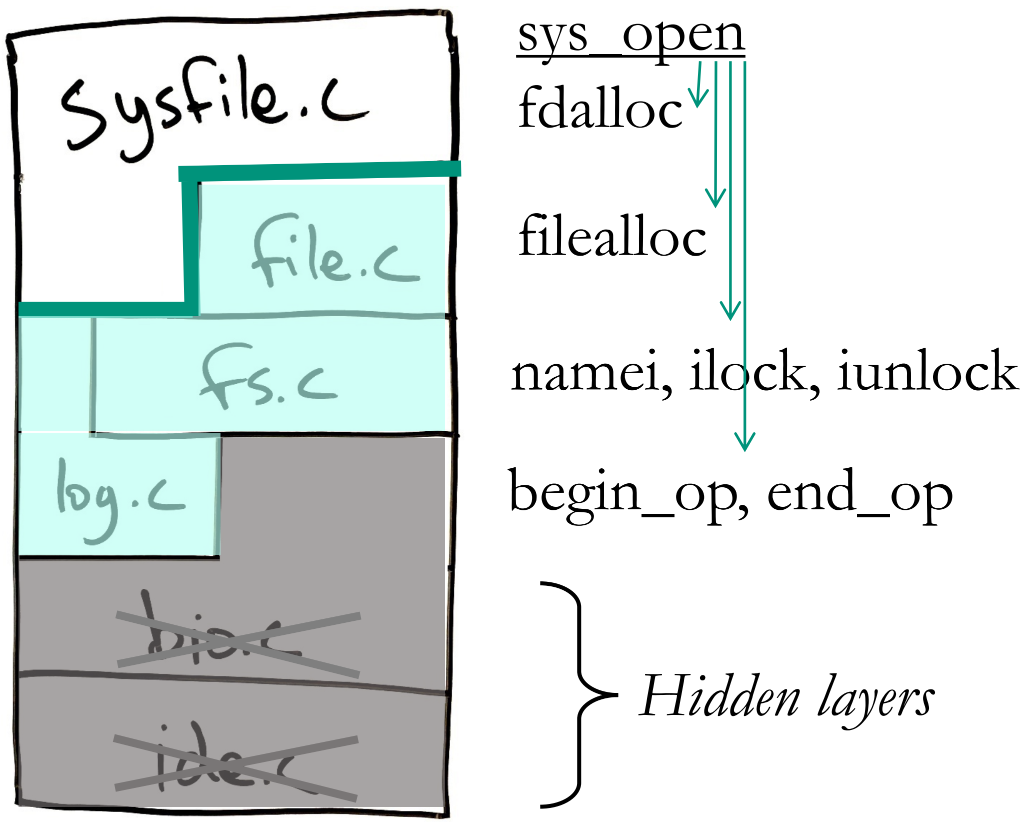
Intermission

"The gluten's back. And it's pissed."

Example: **open**(char*,...) *in sysfile.c*

Assuming that the file exists,
sys_open will call:

- **begin_op**: start a transaction
- **namei**: file path → inode
- **ilock**: lock inode
- **filealloc**: make an empty file struct
- **fdalloc**: list the file among process' open files & get fd #
- **iunlock**
- Store inode # in file struct.
- **end_op**: end a transaction

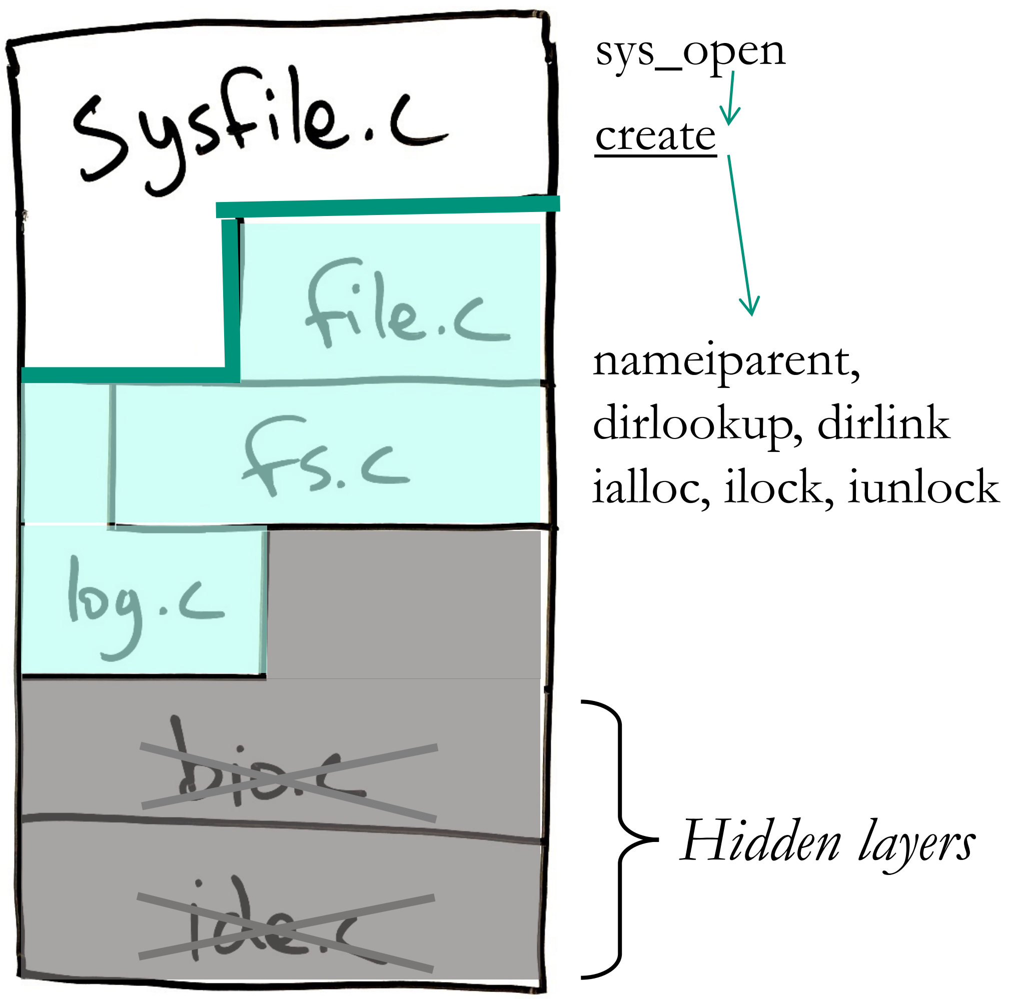


sys_open implementation interacts with three layers below, but none further down.

Example: `create(char*,...)` *in sysfile.c*

If we tell `sys_open` to create a new file, `create` will call:

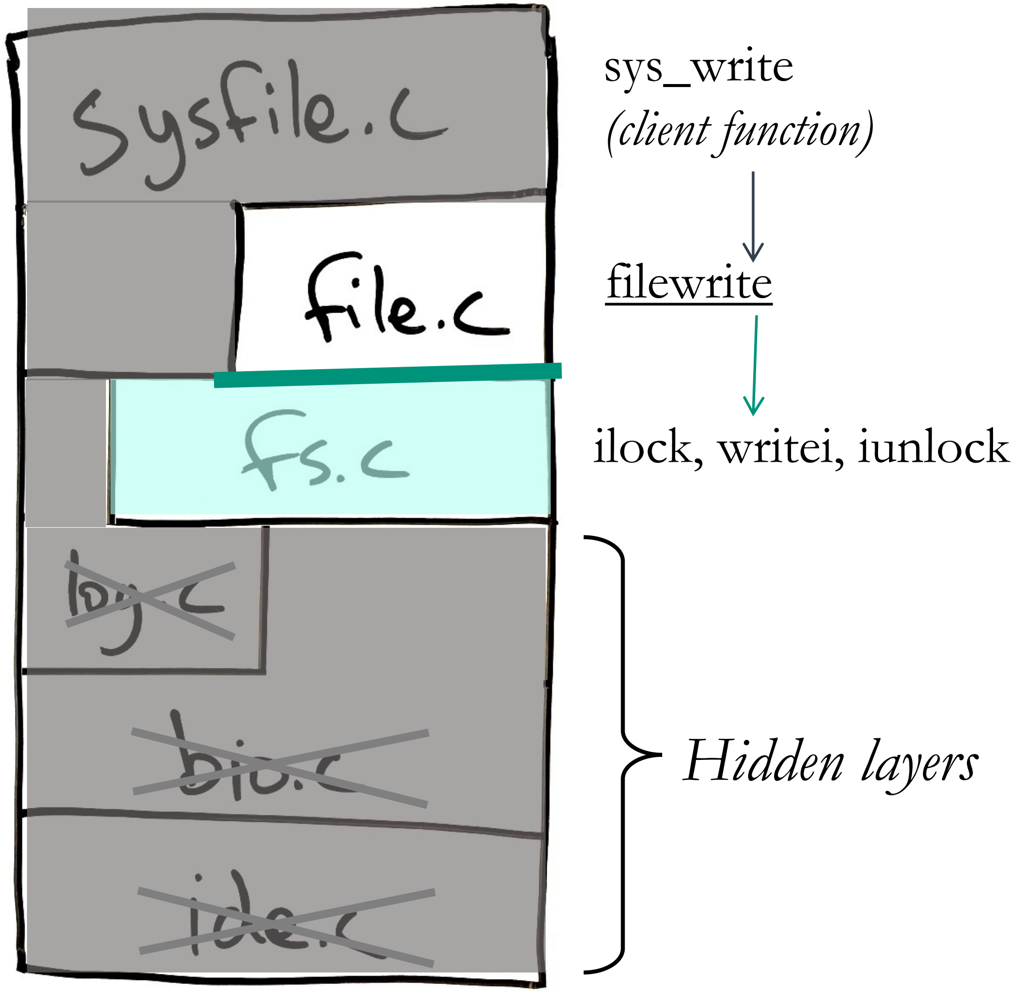
- `nameiparent`: file path → parent inode
- `ilock`: lock parent inode
- `dirlookup`: check whether file exists
- `ialloc`: get inode for new file
    - `ilock` it
    - Set device number and nlink=1
- `dirlink`: add new file to parent directory inode
- `iupdate` new & parent inodes
- `iunlock`: new & parent inodes



Example: filewrite(File*,char *,int n) *in file.c*

sys_write calls `begin_op`, finds the appropriate `struct file` and calls **filewrite**, which calls:

- **ilock**
- **writei**: the inode number and offset both come from `struct file`.
- **iunlock**

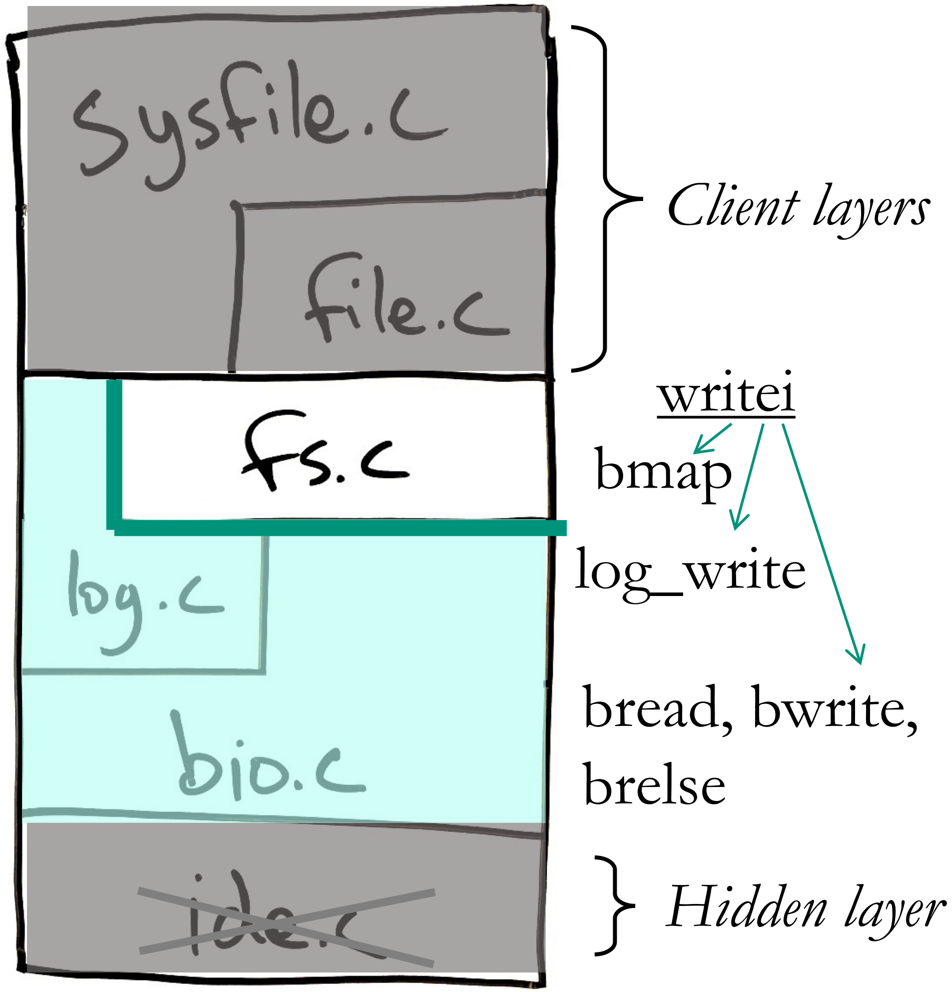


Example: **`writei`**`(Inode*,char *, int offset, int n)` *in fs.c*

Now we're working in a lower layer:

- **`bmap`** : examine the inode to determine which block number corresponds to the write offset.
- **`bread`**: to get the appropriate buffer
- **`bwrite`**: marks the buffer as written
- **`log_write`**: adds it to the transaction
- **`brelse`**: release the buffer lock

Notice that `bread` and `brelse` simultaneously handle locking & buffer memory management.



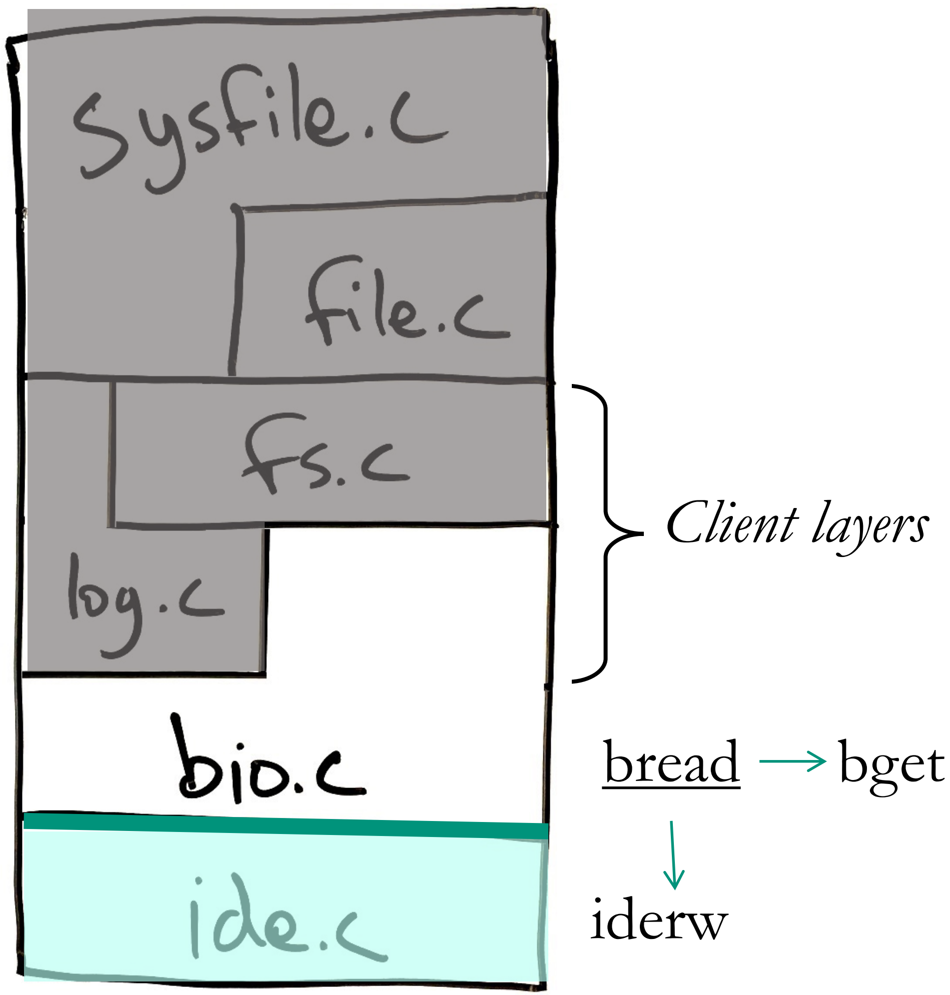
Example: **`bread`**`(int device, int sector)` *in bio.c*

Now we're down at the buffer layer.

- **`bget`**`(…, int sector)`: gets a cached copy or a fresh, empty buffer.
  - In either case, lock the buffer.

If we missed in the cache, call:

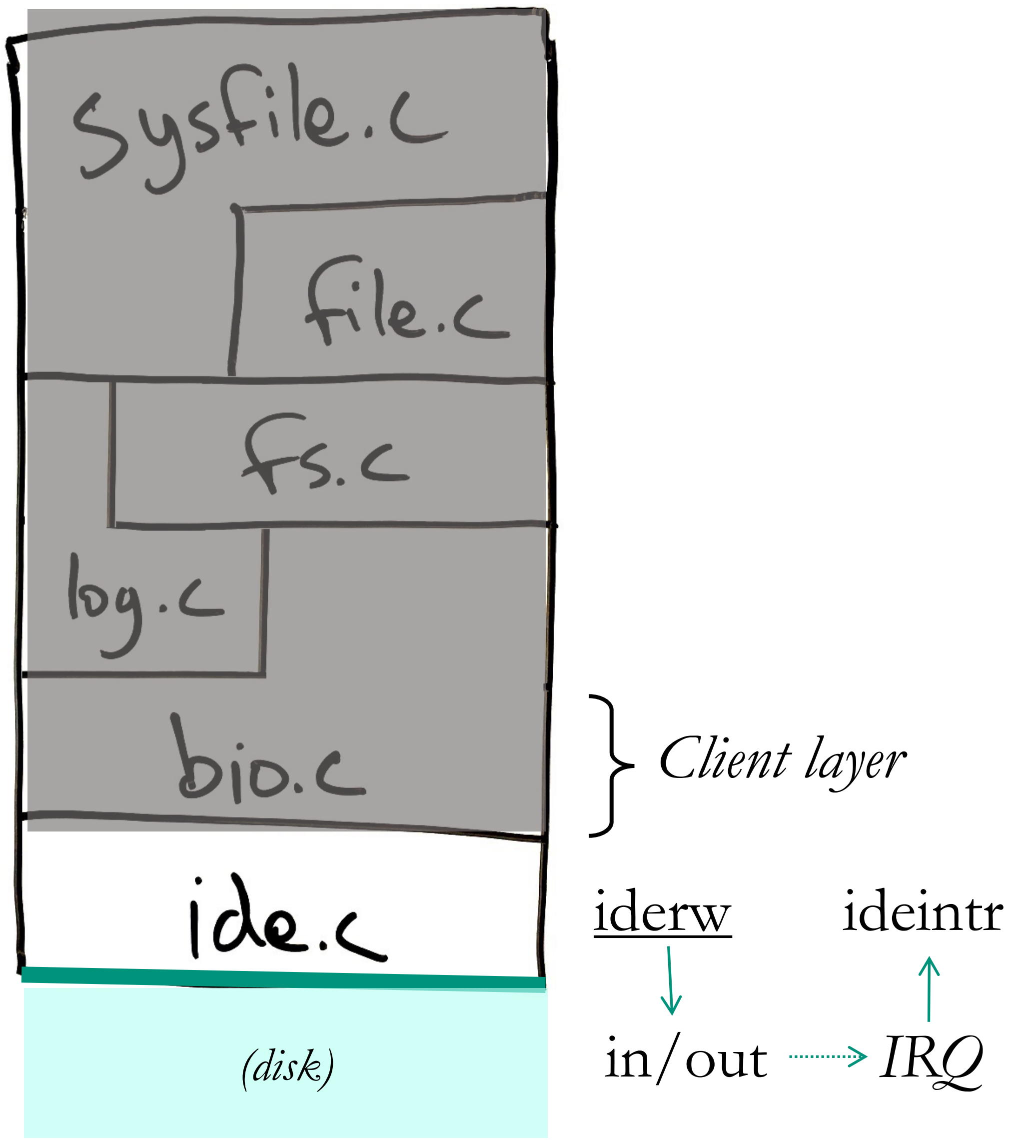
- **`iderw`**: to read the sector from disk.



Example: `iderw(struct buf*)` *in ide.c*

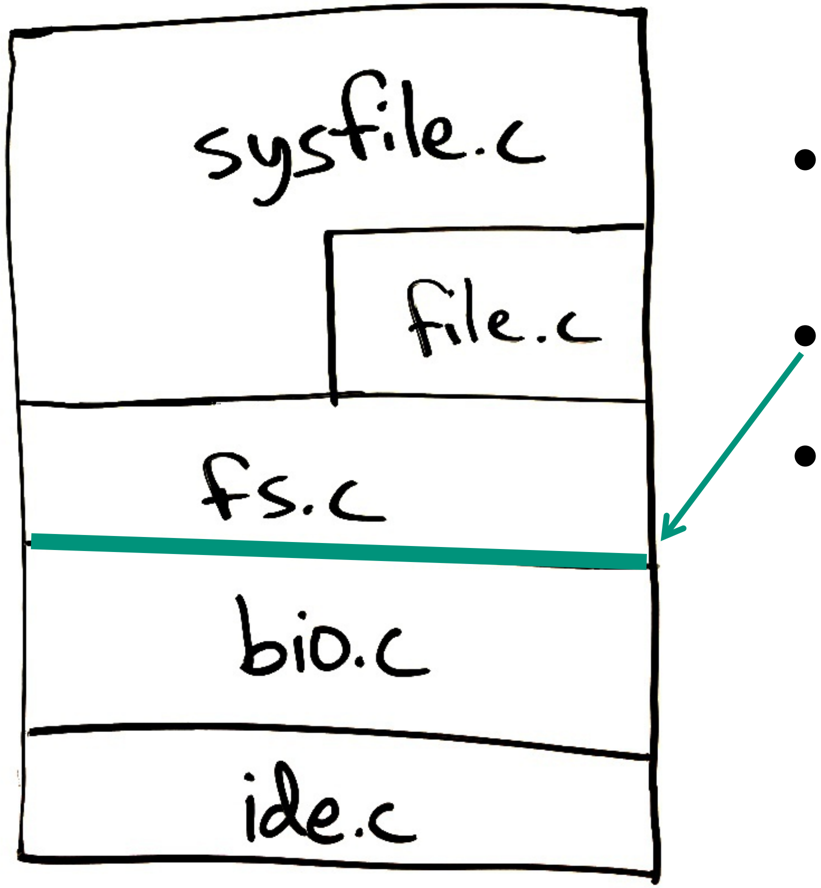
This is the lowest level (device driver).

- Uses **`in`**/**`out`** assembly instructions and port numbers corresponding to device registers.
- Implements an interrupt handler function, **`ideintr`**, which wakes the caller.



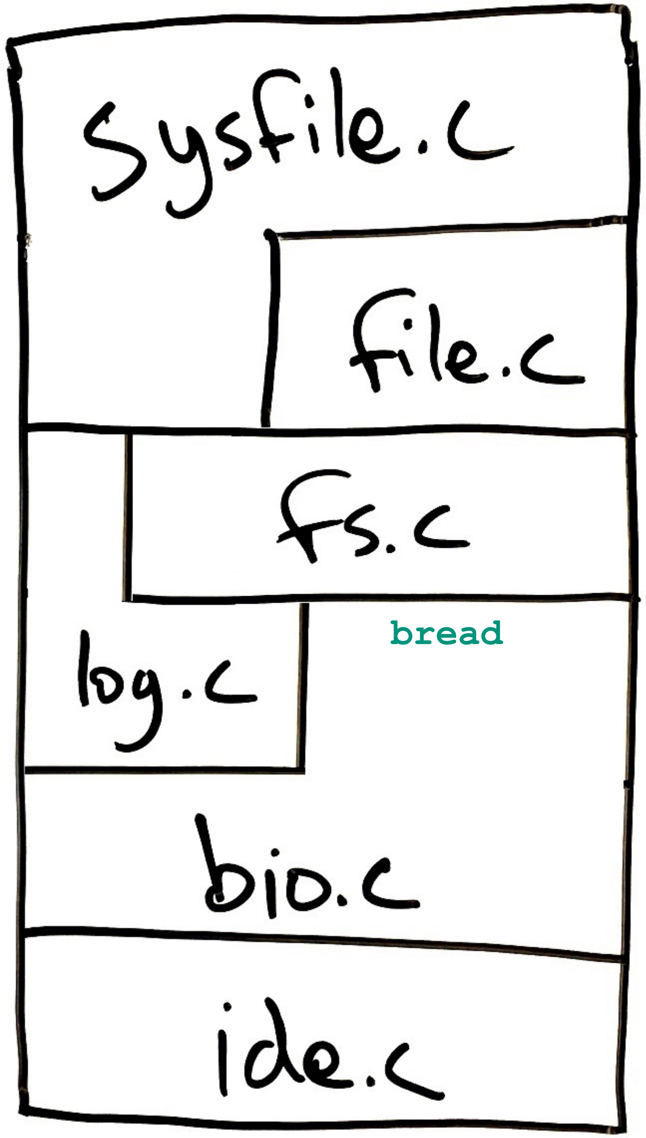
Adding logging to xv6

Old xv6 storage stack



- At some point, Prof. Kaashoek at MIT decided to add write-ahead-logging to xv6.
- This was actually easy to do (~300 LoC)
  https://github.com/mit-pdos/xv6-public/commit/13a96bae
- Just added a layer between fs.c & bio.c
- Implementation of fs.c would have to change, but we can minimize this by keeping the interface mostly the same:
  - Try to provide an API similar to: `bread, bwrite, brelse`.

xv6 logging implementation is a *partial* layer



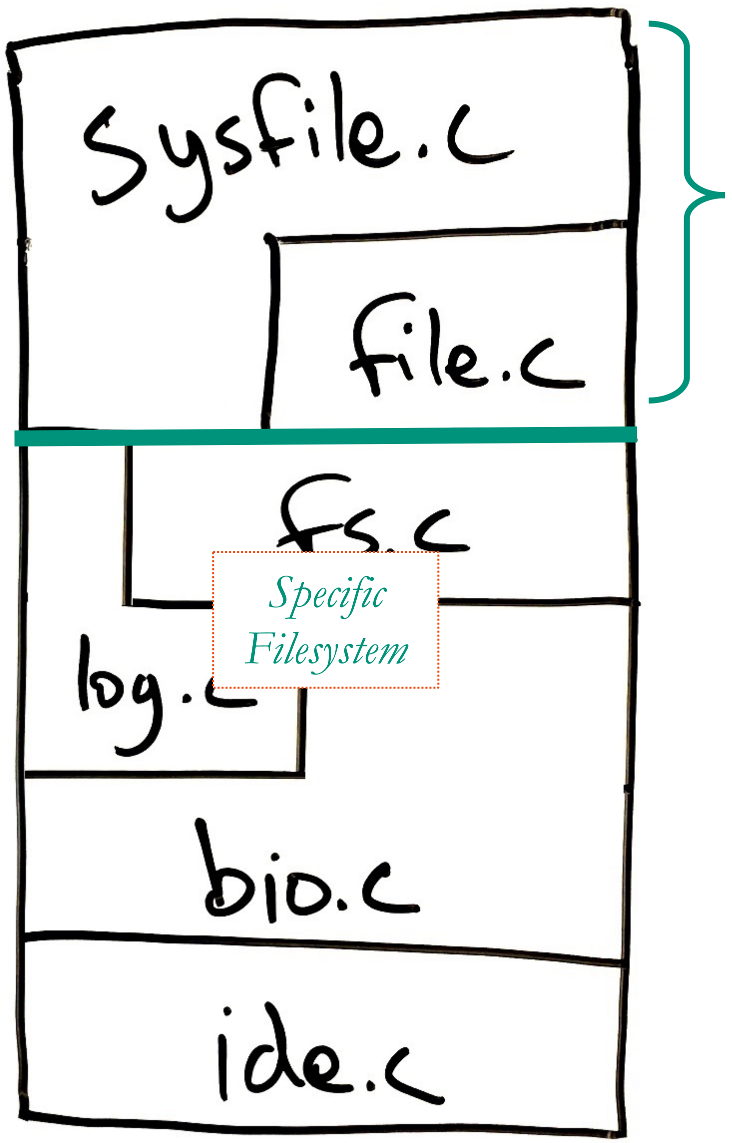
First version of logging API was:

- **`begin_trans`** to start a transaction
- **`log_write`** for writes within a transaction
- **`commit_trans`** to end a transaction

- A *read* function was not included, so client layers must call **`bread`** in lower layer.
- Change *bwrite* → *log_write* in fs.c
- Call *begin_trans* whenever entering a file-related syscall.
- Call *commit_trans* when entering any other syscall

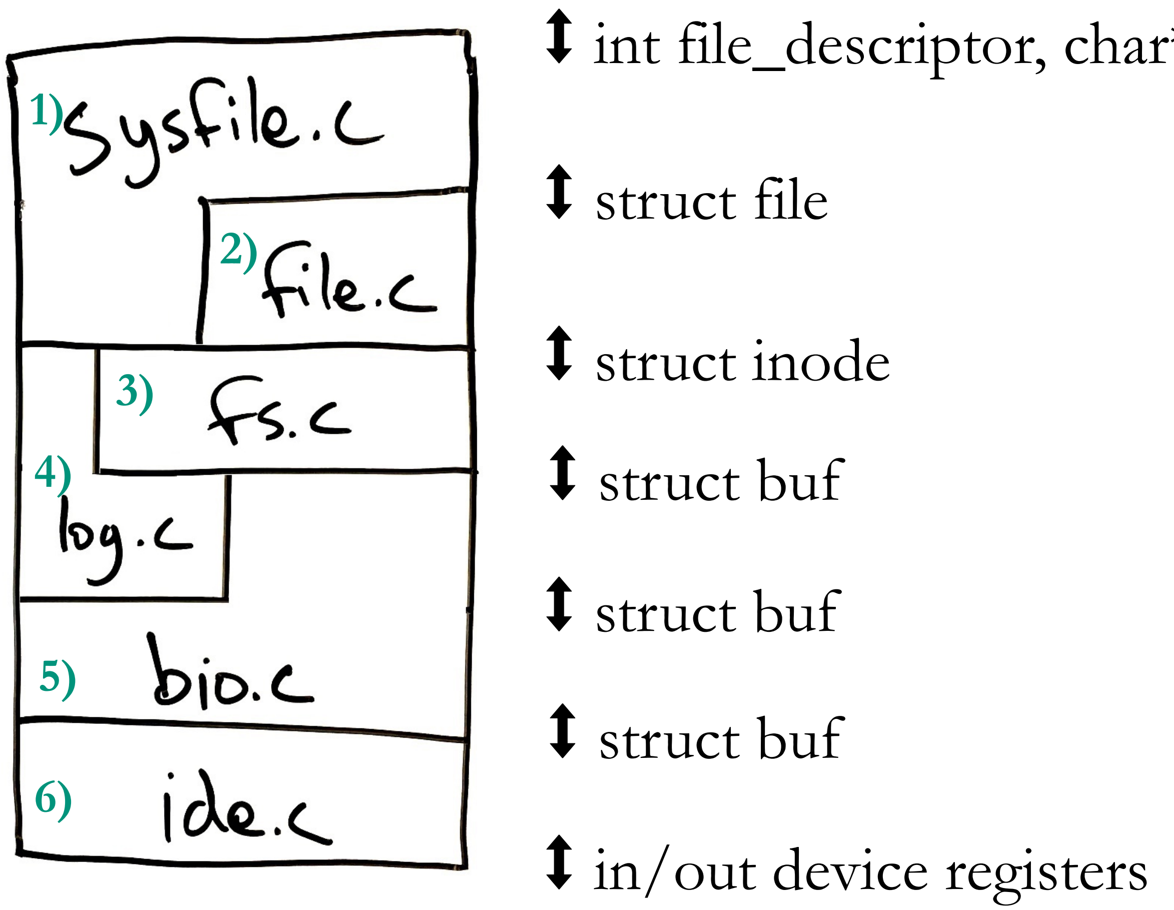
- Later, the log was allowed to store multiple transactions & functions were renamed to **`begin_op`**, **`end_op`**
  https://github.com/mit-pdos/xv6-public/commit/71453f72f2

Supporting multiple filesystems



- xv6 supports just one filesystem
- Syscall implementations work directly with inodes (a filesystem detail).
  - But some filesystems don't even use inodes!
  - Logging implementation is also specific to FS
- Linux has a ***virtual file system*** (VFS) layer to support multiple FSs.
- xv6's lack of strict layering leads to less code, but it's less *extensible*.

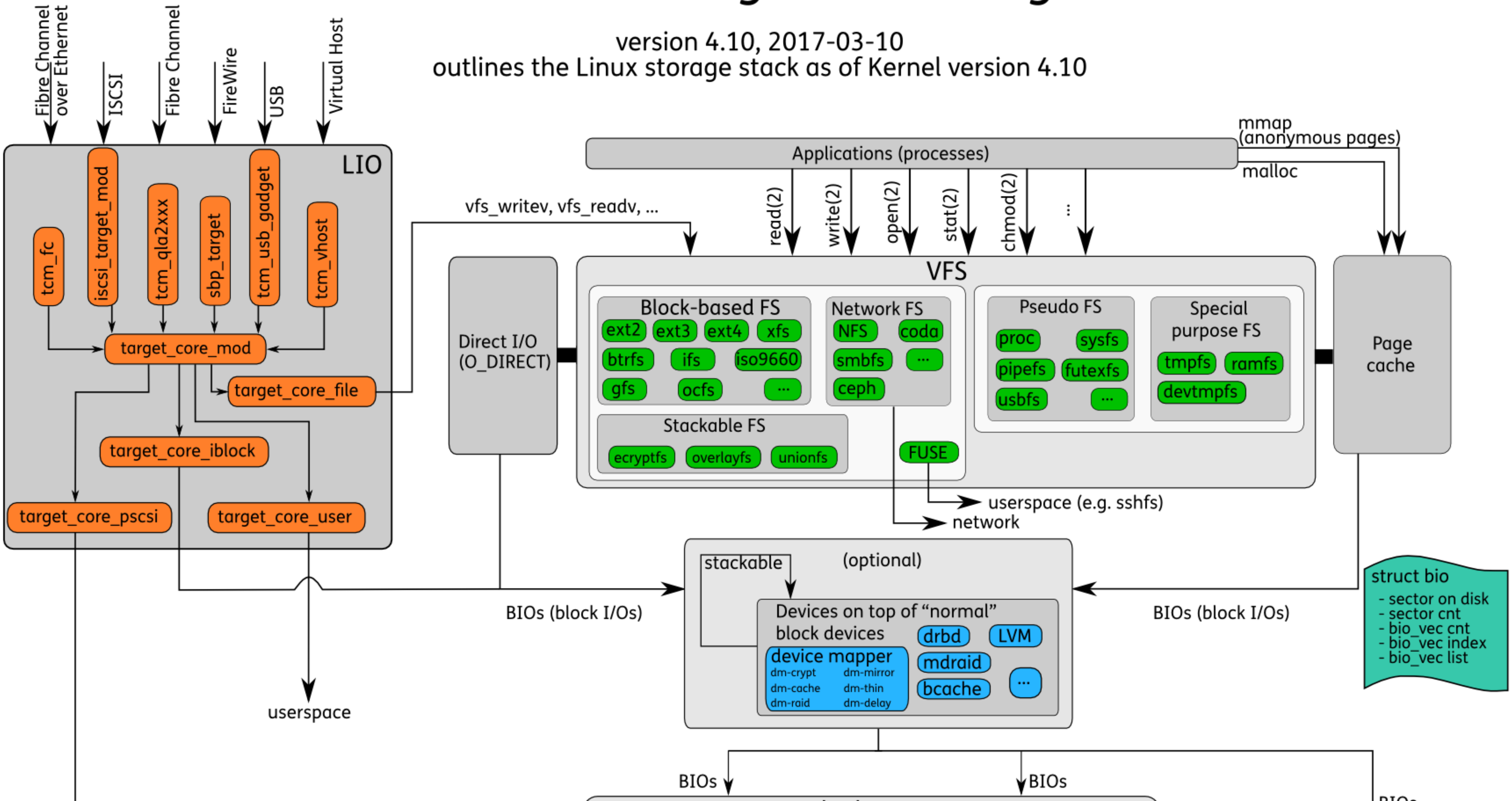
Where would you *insert* a software RAID layer?

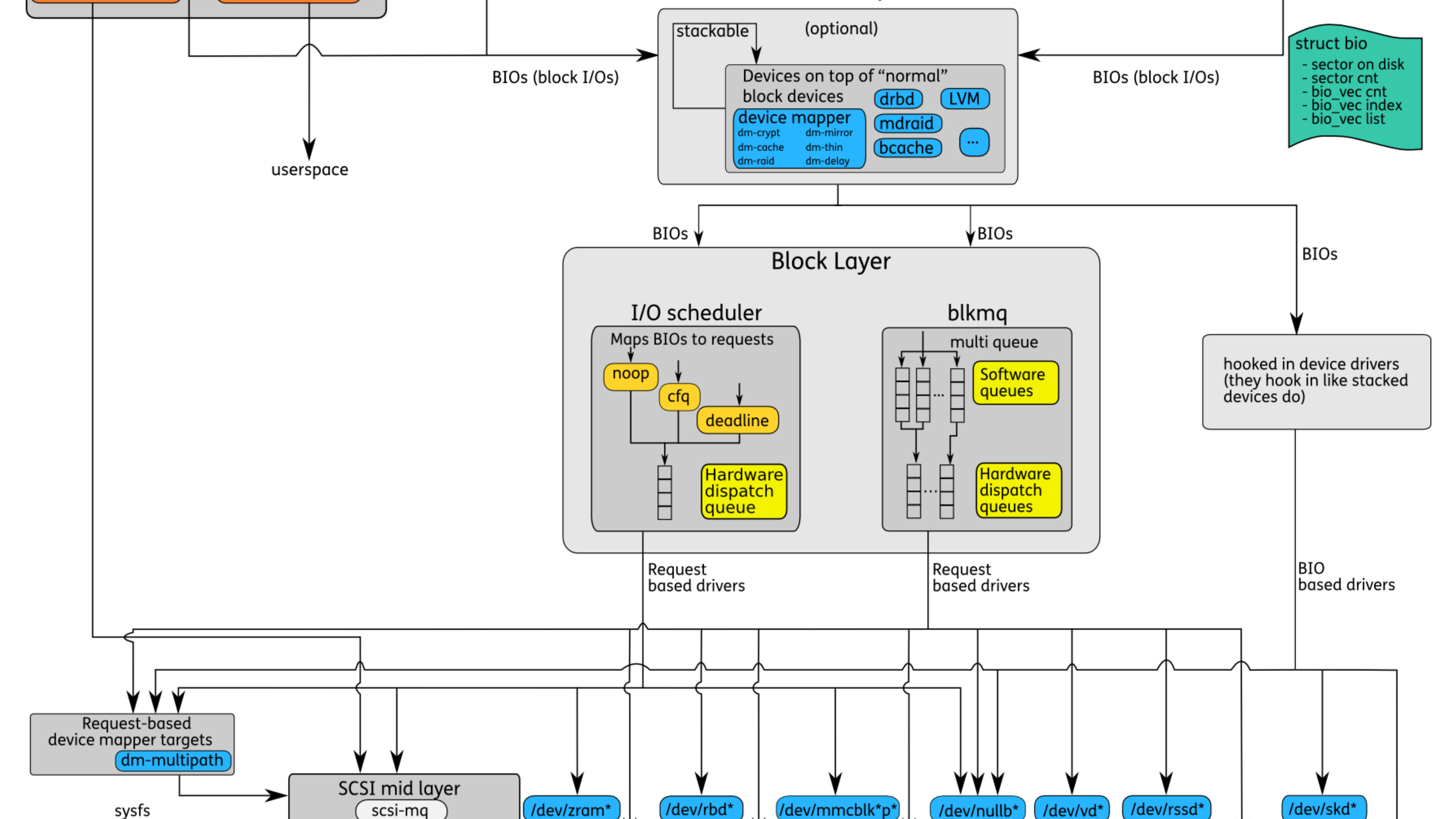


- Somewhere below layers 3, 4, or 5. Below bio.c would be cleanest.
- *Hardware* RAID would be below this kernel software stack, perhaps with a new device driver (replacing ide.c)

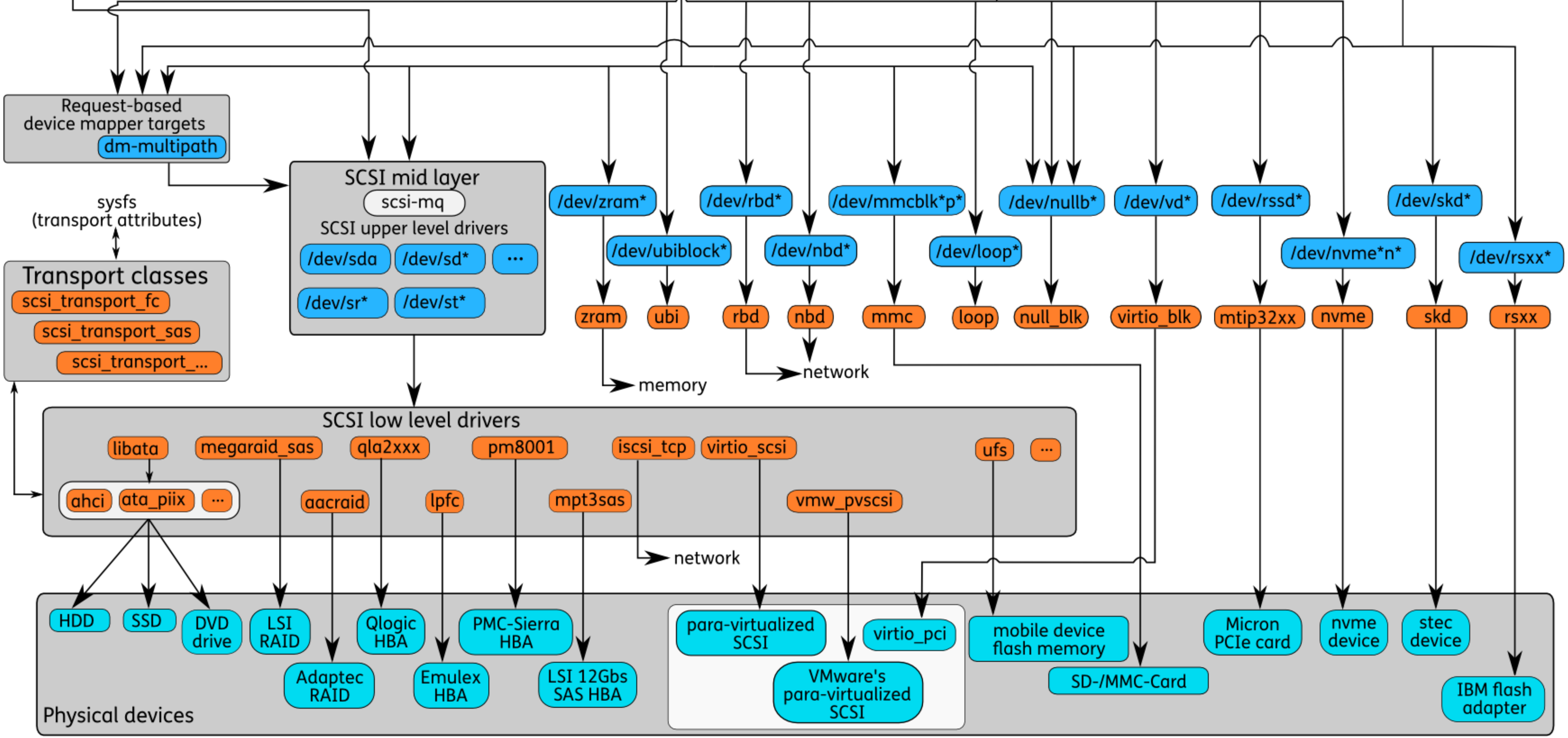
The Linux Storage Stack Diagram

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





stackable
(optional)
Devices on top of "normal" block devices
drbd
LVM
device mapper
dm-crypt
dm-mirror
dm-cache
dm-thin
dm-raid
dm-delay
mdraid
bcache
...
struct bio
- sector on disk
- sector cnt
- bio_vec cnt
- bio_vec index
- bio_vec list
BIOs (block I/Os)
BIOs (block I/Os)
userspace
BIOs
BIOs
BIOs
Block Layer
I/O scheduler
Maps BIOs to requests
noop
cfq
deadline
Hardware dispatch queue
blkmq
multi queue
Software queues
Hardware dispatch queues
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
SCSI mid layer
scsi-mq
sysfs
/dev/zram*
/dev/rbd*
/dev/mmcblk*p*
/dev/nullb*
/dev/vd*
/dev/rssd*
/dev/skd*



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

Recap – Storage Layer Interactions

- Showed layered design of xv6 storage system
- Implementation of each layer uses only the layer(s) directly below
  - Must provide an API suitable for implementing the layer(s) directly above
  - Deeper layer are hidden.
- **`defs.h`** makes a subset of kernel functions in each file "public."
- Linux has a virtual file system (VFS) layer that allows multiple filesystems to coexist in one machine.