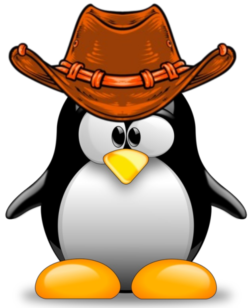
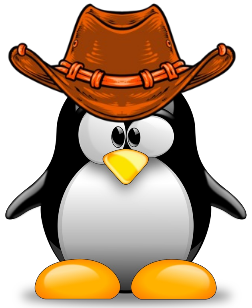


CSE 330:
Operating Systems

Adil Ahmad

Lecture #17: The block abstraction and the file system



Recap of last week

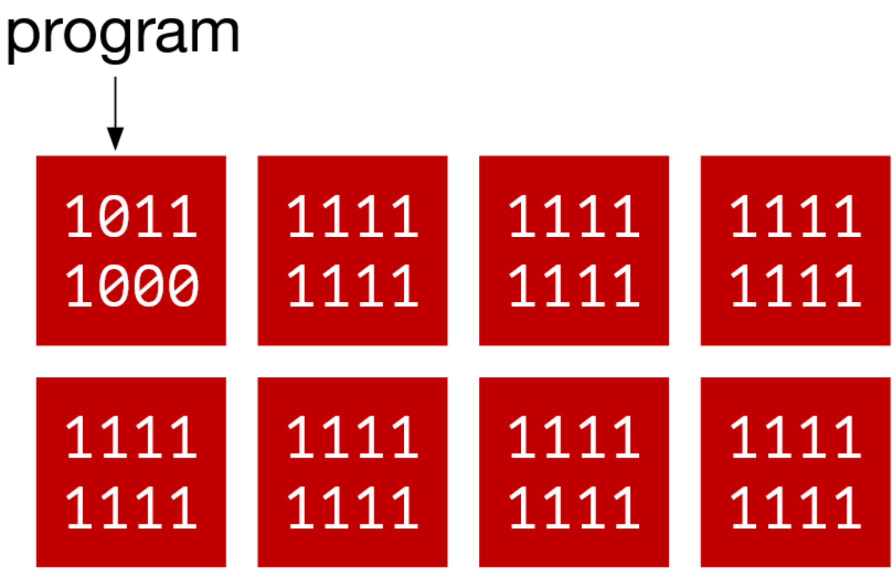
Writing to a flash region

- All “1” in a block → block is **clean** and ready to be used
- Data is **written** by changing the “1” into a “0” (called **program**)
  - Operation at “page-level”
- To remove data, you must write back “1” (called **erase**)
  - Operation at “block-level” (block → a set of pages)
- Hence, program is much faster than erase

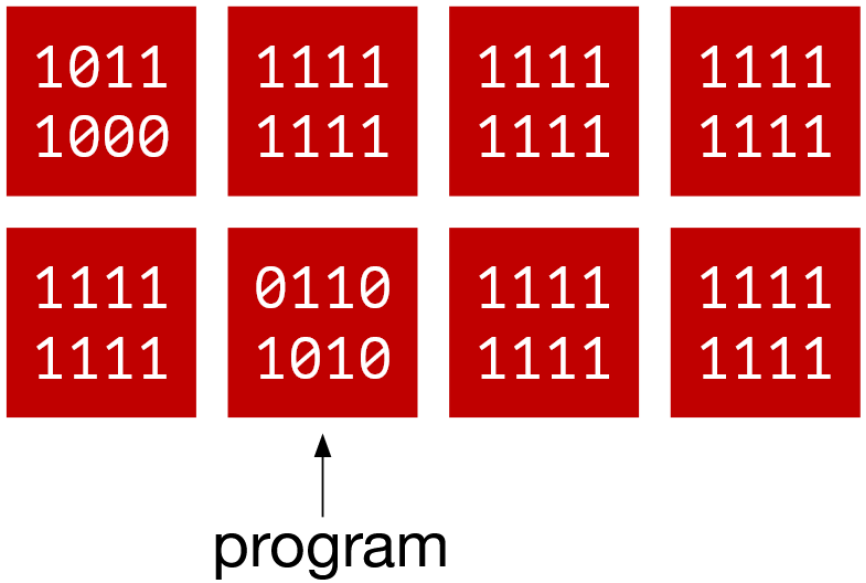
Writing to a flash region (illustration)

All pages are clean
("programmable")

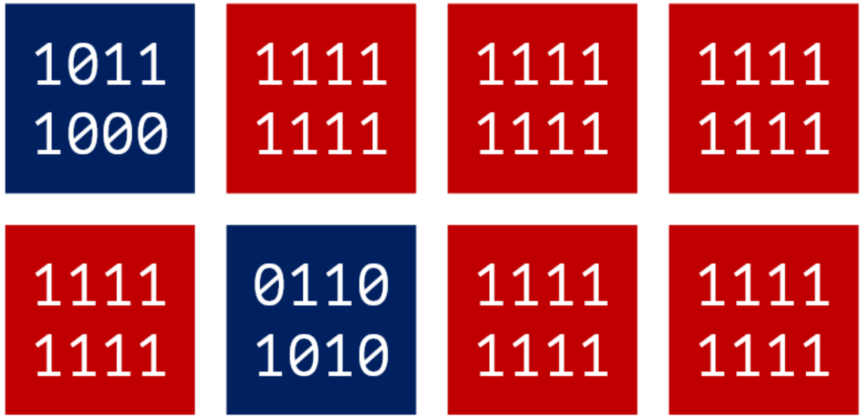
Writing to a flash region (illustration)



Writing to a flash region (illustration)

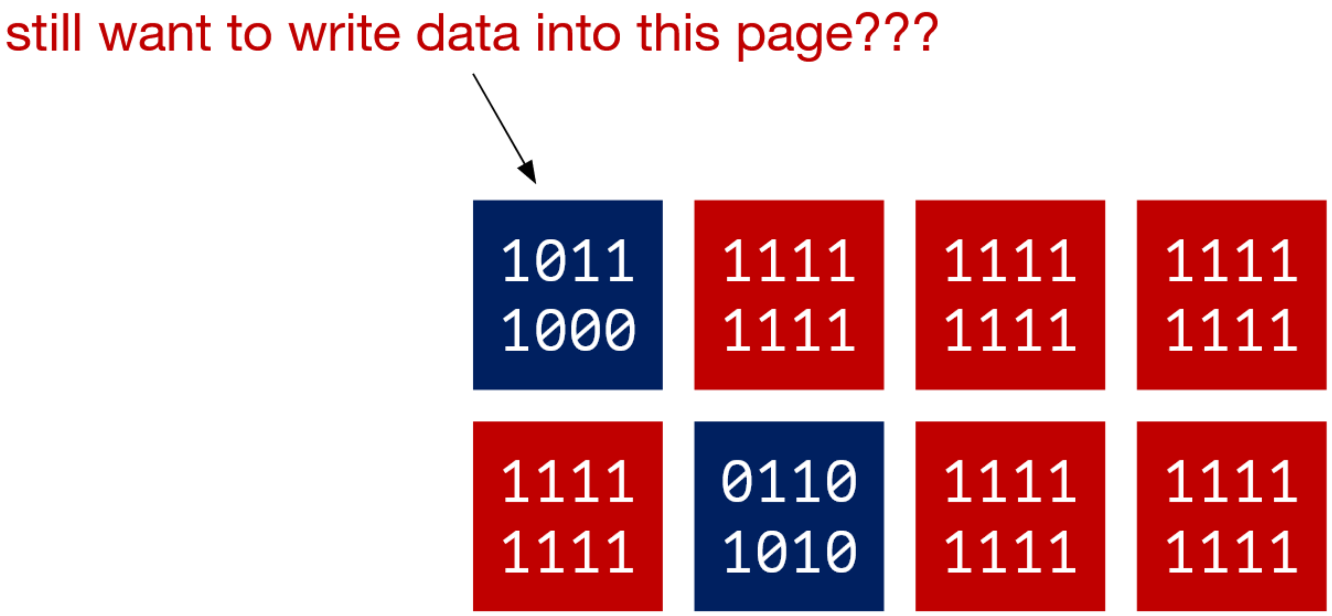


Writing to a flash region (illustration)



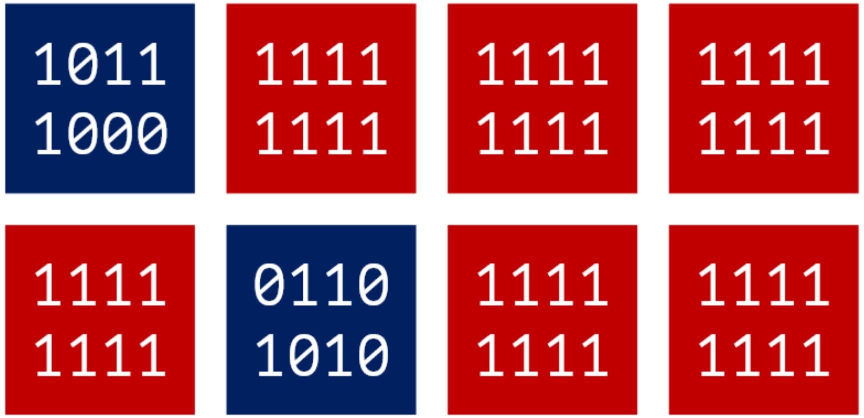
Two pages hold data
(**cannot be overwritten**)

Writing to a flash region (illustration)



Two pages hold data
(**cannot be overwritten**)

Writing to a flash region (illustration)



erase

Writing to a flash region (illustration)

erase
(the whole block)

Writing to a flash region (illustration)

After erase, again, **free state**
(can write new data in any page)

Writing to a flash region (illustration)

This blue page holds data

Comparison of APIs OS must support for HDDs and SSDs

	disk	flash
read	read sector	read page
write	write sector	**program** page (0's) **erase** block (1's)

Can anyone tell me a problem with having different interfaces?

	disk	flash
read	read sector	read page
write	write sector	**program** page (0's) **erase** block (1's)

- Each new storage disk could have its own R/W mechanism
- OS developers would have to cater individually to the specific APIs of a device

This adds **complexity** to using disks for any general task (e.g., writing a file system)

What’s the solution to this “many interface” problem?

	disk	flash
read	read sector	read page
write	write sector	**program** page (0’s) **erase** block (1’s)

Create a **unified interface** and translate all operations from different interfaces to the unified one

Visualizing a **unified interface** for storage access

	disk	flash
read	read sector	read page
write	write sector	**program** page (0's) **erase** block (1's)

block_read(*device*, *location*, *data*)

```
hdd_write:
        seek();
        rotate();
        transfer();
```

```
ssd_write:
        If (erase) erase_block();
        program(page, data);
```

block_write(*device*, *block/sector*, *data*)

The block abstraction layer

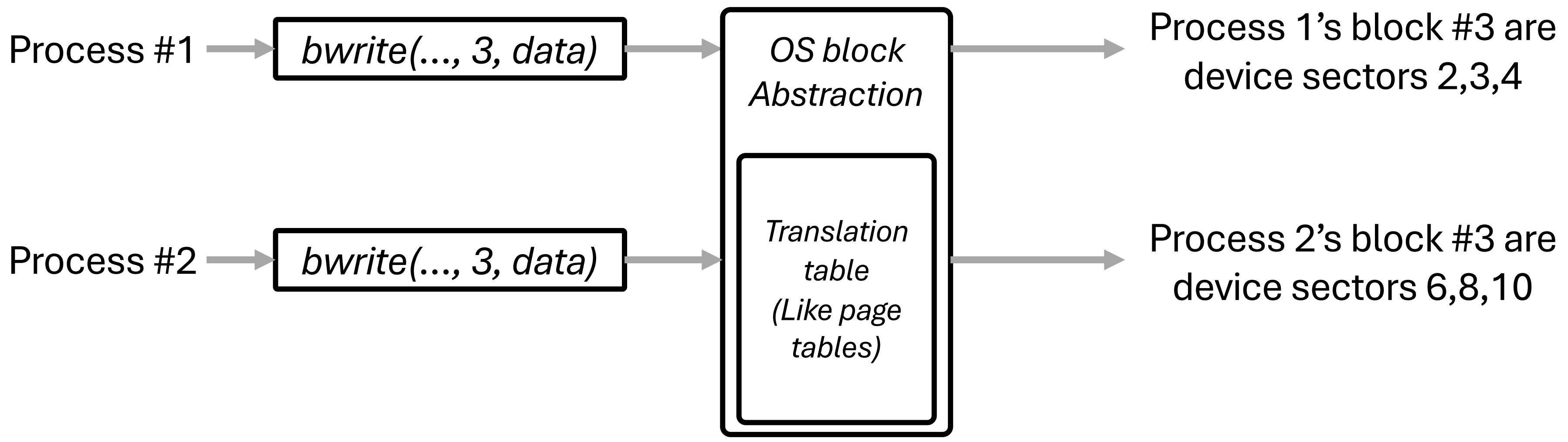
- We want storage devices to allow **random** access of sectors or *blocks* (typically 512 or 4096 bytes)

- Simplify accessing all these devices with a block-level abstraction
  - Higher-level OS code can directly interact with blocks, and the lower-level device driver can support that interaction

- Generally, two operations:
  - bread(device, x, buffer) → read from device at sector X into buffer
  - bwrite(device, y, buffer) → write buffer into sector Y of device

Apart from simplicity, another benefit of block abstraction?

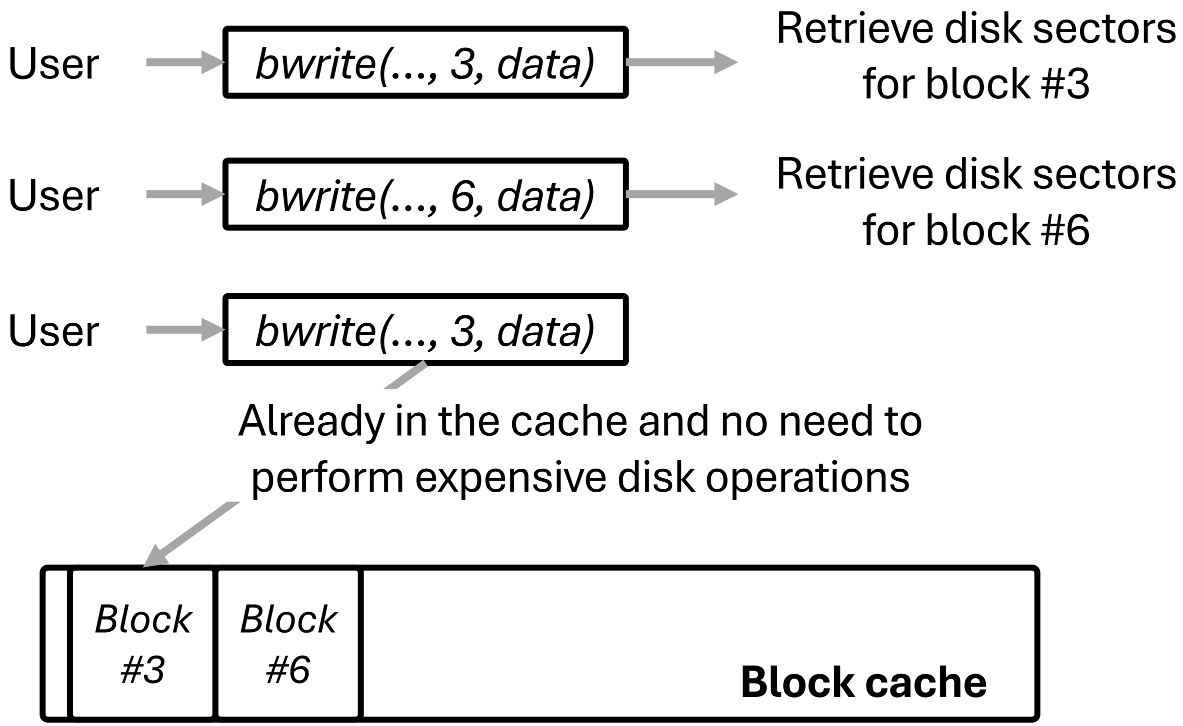
- **Logical disk view:** OS can virtualize how a user thinks of disk regions, and even combine disk "sectors" into larger "blocks"



- Multiple users/processes can simply operate on block #1 → #N

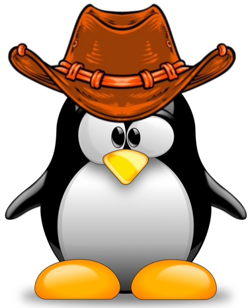
Apart from simplicity, another benefit of block abstraction?

- **Caching efficiency:** Conveniently keep track of frequently-used disk regions by users and keep them in-memory for later use



Are there any disadvantages of the block abstractions?

- **Sacrifice raw device I/O performance**
  - Each layer of abstraction requires translation → many more function calls and memory accesses
  - Much faster to directly access block-level storage for special scenarios (e.g., where disk writes *are* needed)



Let’s review a block layer implementation

High-level working of the **xv6 OS** block layer

- **Step #1:** Initializes a list of in-memory block cache
- **Step #2:** Reads data from disk into an in-memory block cache, while synchronizing block access with other threads
- **Step #3:** Performs a write (if needed) on an in-memory block
  - Similar synchronization as step #2

Block layer initialization (xv6/kernel/bio.c)

```c
struct {
  struct spinlock lock;
  struct buf buf[NBUF];

  // Linked list of all buffers, through prev/next.
  // Sorted by how recently the buffer was used.
  // head.next is most recent, head.prev is least.
  struct buf head;
} bcache;

void
binit(void)
{
  struct buf *b;

  initlock(&bcache.lock, "bcache");

  // Create linked list of buffers
  bcache.head.prev = &bcache.head;
  bcache.head.next = &bcache.head;
  for(b = bcache.buf; b < bcache.buf+NBUF; b++){
    b->next = bcache.head.next;
    b->prev = &bcache.head;
    initsleeplock(&b->lock, "buffer");
    bcache.head.next->prev = b;
    bcache.head.next = b;
  }
}
```

Synchronize using a lock

Why use spinlock here?

Linked list of blocks to cache, and keep *head*

Initialize the spinlock

Why use sleeplock here?

Initialize the sleeplock

Read a block from disk (xv6/kernel/bio.c)

```c
// Return a locked buf with the contents of the indicated block.
struct buf*
bread(uint dev, uint blockno)
{
  struct buf *b;

  b = bget(dev, blockno);
  if(!b->valid) {
    virtio_disk_rw(b, 0);
    b->valid = 1;
  }
  return b;
}
```

Get a free block

If block is empty, fetch from disk

```c
// Look through buffer cache for block on device dev.
// If not found, allocate a buffer.
// In either case, return locked buffer.
static struct buf*
bget(uint dev, uint blockno)
{
  struct buf *b;

  acquire(&bcache.lock);

  // Is the block already cached?
  for(b = bcache.head.next; b != &bcache.head; b = b->next){
    if(b->dev == dev && b->blockno == blockno){
      b->refcnt++;
      release(&bcache.lock);
      acquiresleep(&b->lock);
      return b;
    }
  }

  // Not cached.
  // Recycle the least recently used (LRU) unused buffer.
  for(b = bcache.head.prev; b != &bcache.head; b = b->prev){
    if(b->refcnt == 0) {
      b->dev = dev;
      b->blockno = blockno;
      b->valid = 0;
      b->refcnt = 1;
      release(&bcache.lock);
      acquiresleep(&b->lock);
      return b;
    }
  }
  panic("bget: no buffers");
```

Acquire spinlock

Go through list

Release spl and get sleeplock

Go through list again, but find free block

Write a block from disk (xv6/kernel/bio.c)

bread(..) should always happen first, before writing according to xv6

Why is this the case?

All block operations happen in the “cache” before being written to disk

```
// Write b's contents to disk.  Must be locked.
void
bwrite(struct buf *b)
{
  if(!holdingsleep(&b->lock))
    panic("bwrite");
  virtio_disk_rw(b, 1);
}
```

Make sure lock is being held

Perform the actual write

Finalizing the block-level abstraction

Provides us a way to simplify "direct" writes to different storage devices

HDD → seek, rotate, transfer → bread(dev, block, data);
SSD → program, erase → bread(dev, block, data);

The xv6 block abstraction also implements the following:

- **Synchronization** b/w block-level access of a device to avoid corruptions
- **Caching** of recently-used blocks for improved performance

Is the block abstraction enough for development?

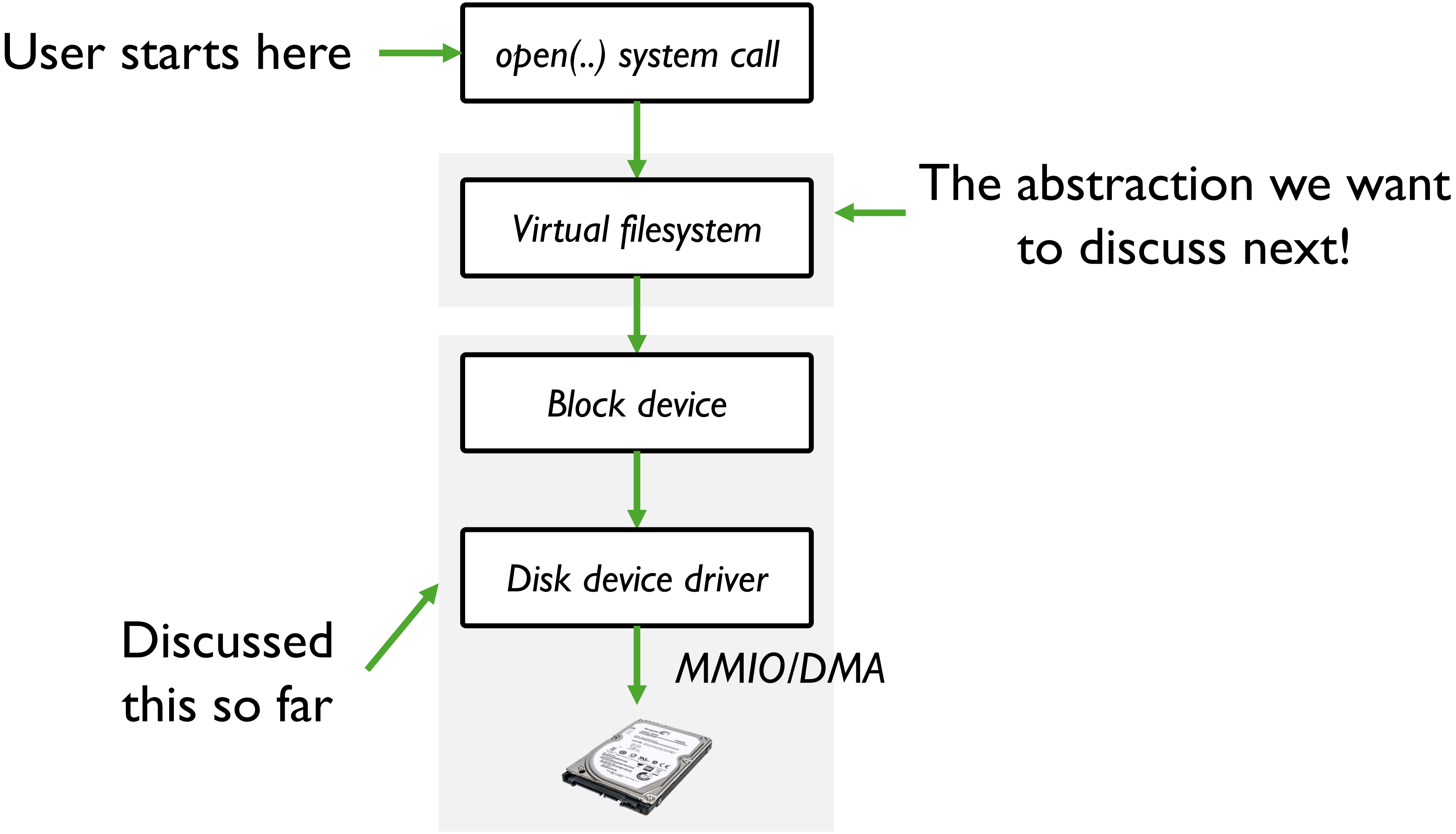
- **No, there are never enough abstractions? ☺**
  - Blocks are too complicated generally for most users
  - Hence, the block abstraction is mostly used by OS developers to build “file systems”

- **Are file systems enough of an abstraction?**
  - No, there are never enough abstractions! ☺

- **What’s an example of an abstraction we build using file systems?**
  - echo “Hello World!” > test.txt (creates and writes to a file)

The (virtual) file system

Disk interactions starting from a system call



The (virtual) filesystem abstraction

- High-level intuitive view of how we look at data stored on disks
- Not just disk-related; UNIX philosophy → "everything is a file"

Can anyone tell me of other ways in which you have used files?

- High-level intuitive view of how we look at data stored on disks
- Not just disk-related; UNIX philosophy → "everything is a file"
  - /dev/memalloc → virtual file to communicate with modules
  - CPU features can be enabled or disabled using files.
    - E.g., entire CPUs can be disabled as follows:

      echo 0 | sudo tee /sys/devices/system/cpu/cpu1/online
  - Perform console R/W → write(1, "hello", 5)
    - 1 → file descriptor for console (terminal) output

Let’s get back to (disk-related) file

- File → a set of blocks that the OS has combined and operates on together
  - Recall that the filesystem is composed of the the block layer

- Files are given identifiers (e.g., “hello.txt” is a human-readable version) so programs/users can distinguish between them

- File system (FS) → an intricate hierarchical collection of files built using the blocks in your storage disk

Different "names" for a file

- Three different names typically
  - ✓ **inode** (low-level names)
    - ➢ Internal name (number) given to a file by the OS
  - ✓ **path** (human readable)
    - ➢ The version that we see when we open the file browser (e.g., Windows explorer or MacOS finder)
  - ✓ **file descriptor** (runtime state)
    - ➢ Represents the runtime status of a certain file

The inode (OS-level representation)

- Each file has exactly one inode number
- Inodes are unique (at a given time) within a FS
- **File names can be the same, why can't inodes?**
  - Something must be unique for the OS to track

PROMPT>: stat test.dat

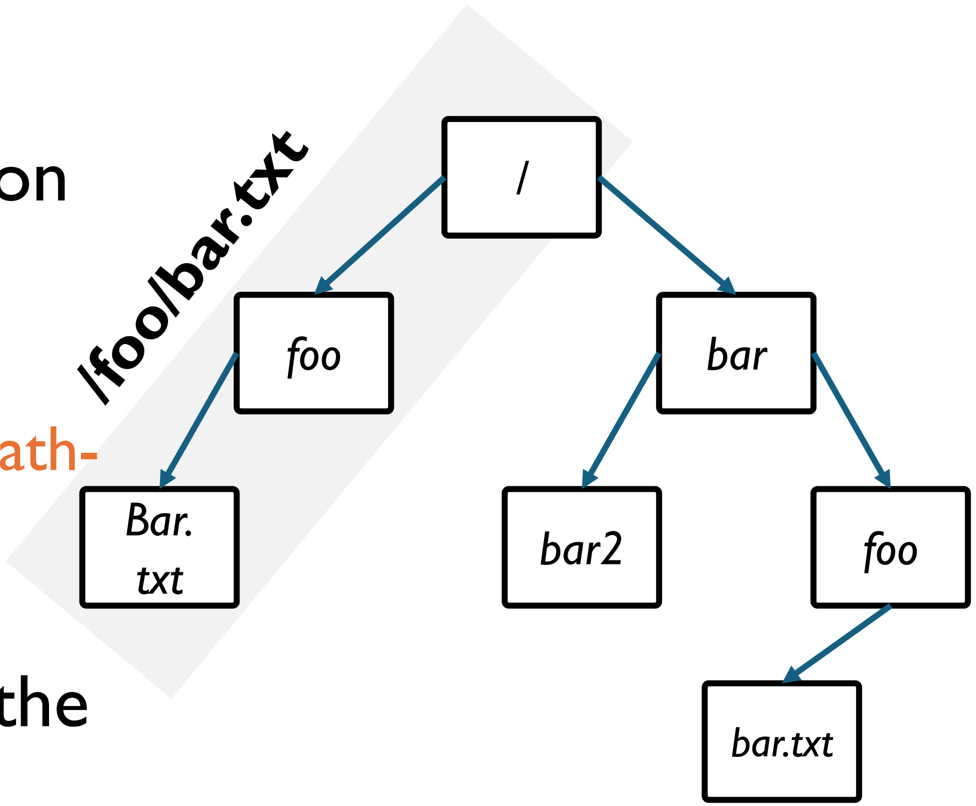
File: 'test.dat'
Size: 5
Blocks: 8
IO Block: 4096 regular file
Device: 803h/2051d
Inode: 119341128
Links: 1
Access: (0664/-rw-rw-r--)
Uid: (1001/ yue)
Gid: (1001/ yue)
Context: unconfined_u:object_r:user_home_t:s0
Access: 2015-12-17 04:12:47.935716294 -0500
Modify: 2014-12-12 19:25:32.669625220 -0500
Change: 2014-12-12 19:25:32.669625220 -0500
Birth: -

`stat` example

```
PROMPT>: stat test.dat
    File: ‘test.dat’  Size: 5      Blocks: 8      IO Block:
    4096   regular file
    Device: 803h/2051d Inode: 119341128   Links: 1
    Access: (0664/-rw-rw-r--)  Uid: ( 1001/      yue)   Gid: (
    1001/      yue)
    Context: unconfined_u:object_r:user_home_t:s0
    Access: 2015-12-17 04:12:47.935716294 -0500
    Modify: 2014-12-12 19:25:32.669625220 -0500
    Change: 2014-12-12 19:25:32.669625220 -0500
    Birth: -
```

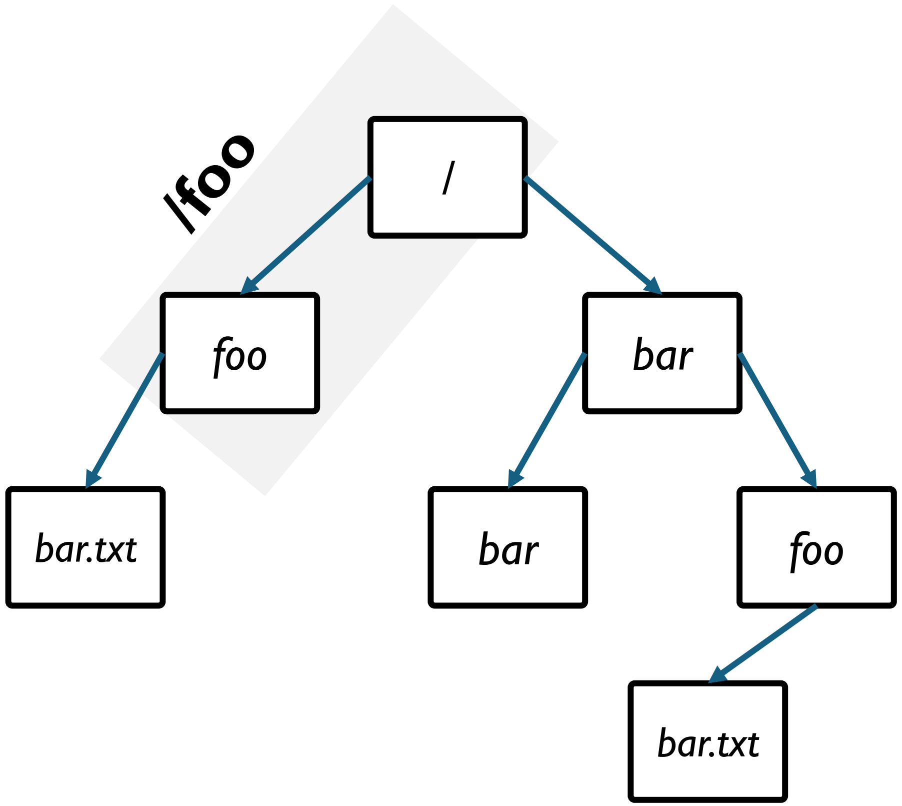
The path (human-readable)

- Human-readable interpretation of every inode
- Typically, represented as – <path-to-directory , filename>
- **Traversing** a tree – getting the final *inode* from a location
  - E.g., ls /foo/bar.txt
  - Gets the inode and prints details



The path (human-readable)

- Directories and files can have the same name if they are in different tree locations
- Cannot have the same name if they're in the same directory
  - ➢ Not really a problem for OSs, but a problem for humans to distinguish between ☺



`ls` example

```
prompt> ls -al
    total 216
    drwxr-xr-x   19 yue   staff    646 Nov 23 16:28 .
    drwxr-xr-x+  40 yue   staff   1360 Nov 15 01:41 ..
    -rw-r--r--@   1 yue   staff   1064 Aug 29 21:48 common.h
    -rwxr-xr-x    1 yue   staff   9356 Aug 30 14:03 cpu
    -rw-r--r--@   1 yue   staff    258 Aug 29 21:48 cpu.c
    -rwxr-xr-x    1 yue   staff   9348 Sep  6 12:12 cpu_bound
    -rw-r--r--    1 yue   staff    245 Sep  5 13:10 cpu_bound.c
    …
```

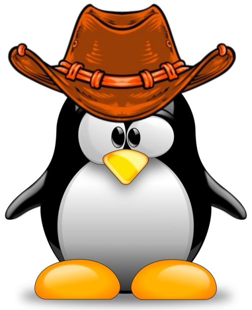
- **Two additional files in every folder. Can anyone tell me what these are?**

The file descriptor

- "Everything is a file"
  - File descriptor tracks what each 'file' really does in the system

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE, FD_DEVICE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe; // FD_PIPE
  struct inode *ip;  // FD_INODE and FD_DEVICE
  uint off;          // FD_INODE
  short major;       // FD_DEVICE
};
```

If file belongs to the disk, it has an inode



Questions? Otherwise, see you next class!