EECS-343 Operating Systems
Lecture 16:
Buffer Caching & Logging

Steve Tarzia

Spring 2019

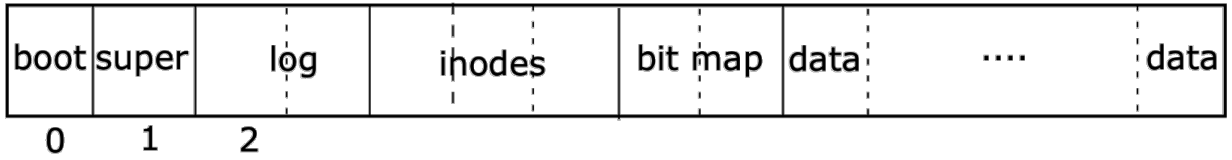
Northwestern

Announcements

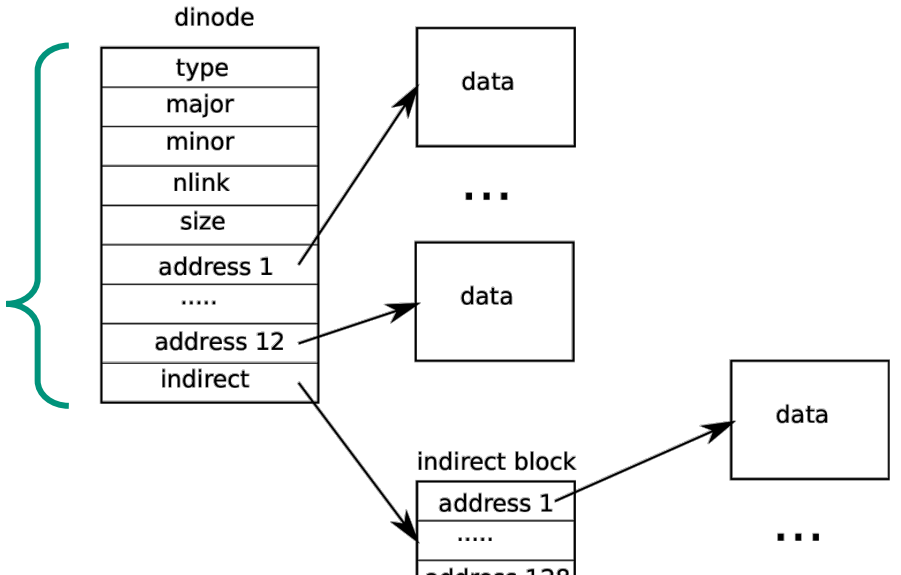
- Project 4 is due next Wednesday
- Homework 4 will be out soon.

Last Lecture – RAID & File Systems

- RAID allows multiple disks to act together for better throughput, capacity, and/or fault tolerance.
  - ***Parity*** is used in ***RAID5*** to achieve all of the above.
- OSes have a application-level API (syscalls) for file I/O:
  - open, read, write, seek, stat, fsync, rename, unlink, mkdir
- ***Filesystem*** is a data structure the OS uses to organize disk space.

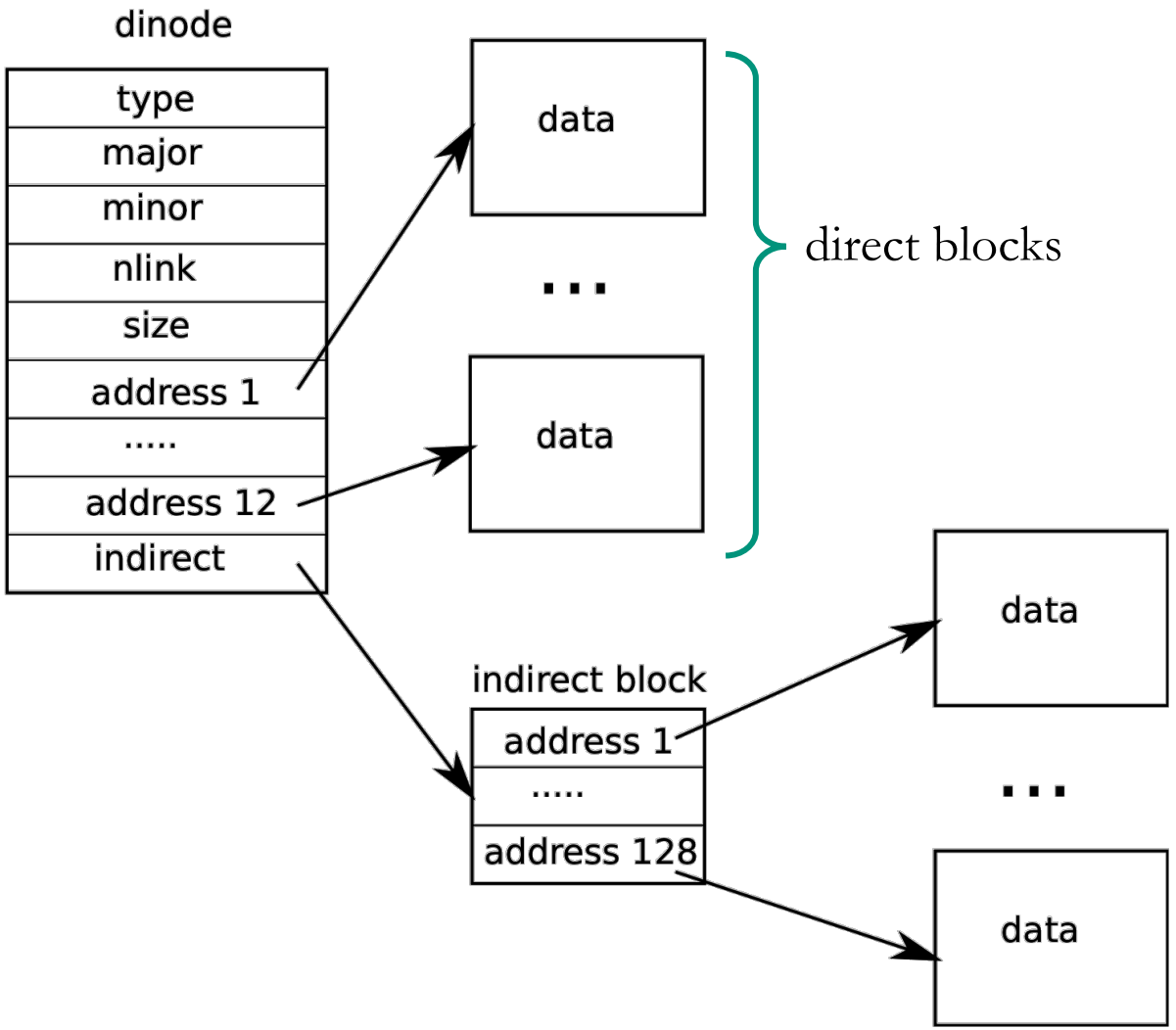


- Each file/directory has an ***inode*** storing metadata & pointers to data blocks.



Inodes (xv6)

- Each file/directory is represented by an inode (`struct dinode`)
- A file inode stores:
  - Reference count (# of hard links)
  - Total file size
  - Array of data blocks storing the file's data (***direct blocks***)
  - Optional ***indirect block*** address, for files larger than 6kB.
- xv6 files can be 70kB at most!
- Inodes are 64 bytes each



Directory inodes (xv6)

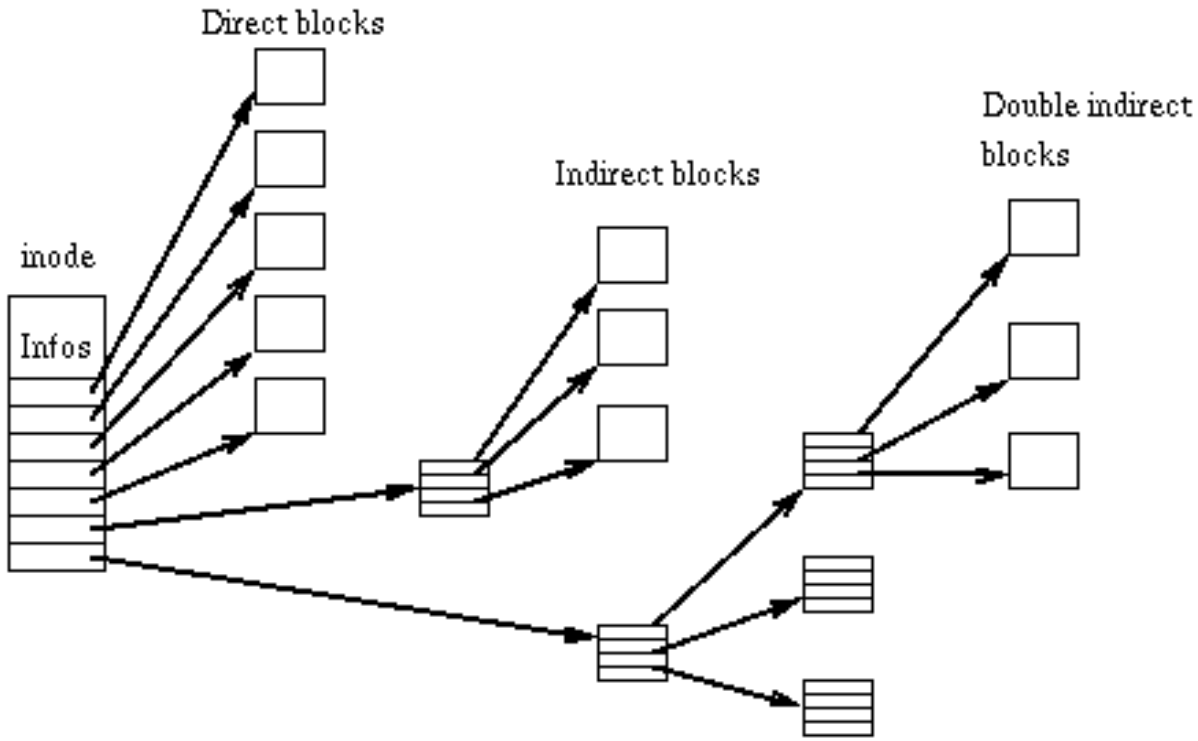
- A Directory is like a file containing an array of <name, inode> pairs:

```
struct dirent {
  ushort inum;
  char name[DIRSIZ]; // 14
};
```

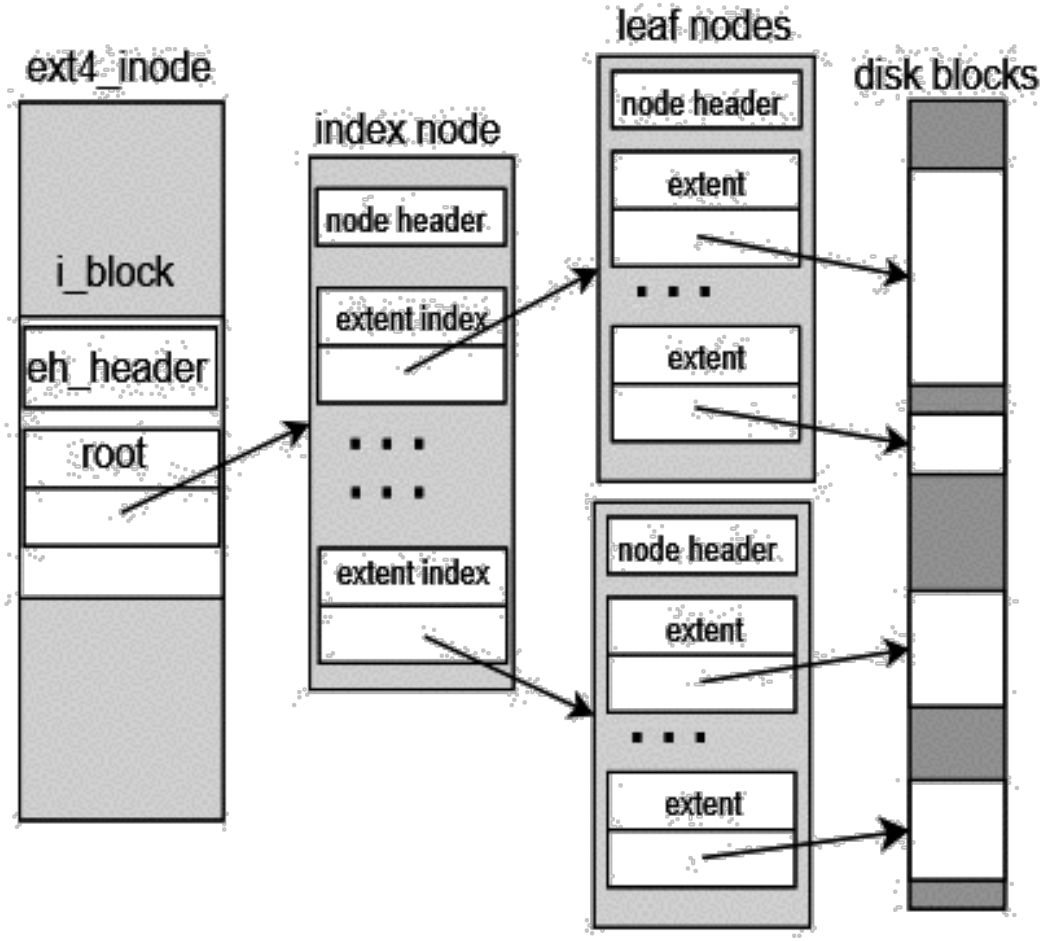
- Inode type is set to T_DIR instead of T_FILE
- Every directory contains two special entries:
  - "`..`" pointing to parent directory
  - "`.`" pointing to self

Storing larger files

ext3: double & *triple* indirect blocks

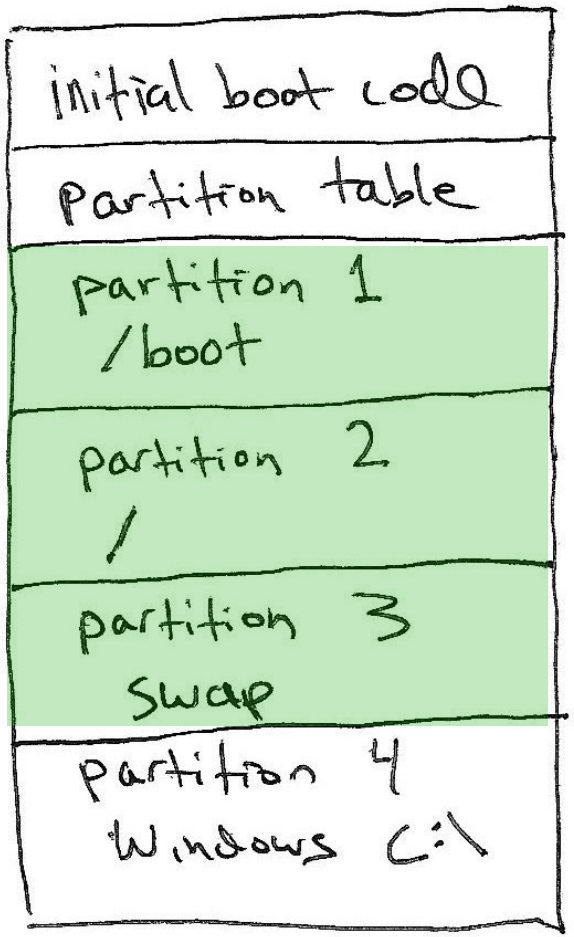


ext4: extents



Disk partitions

Disk A



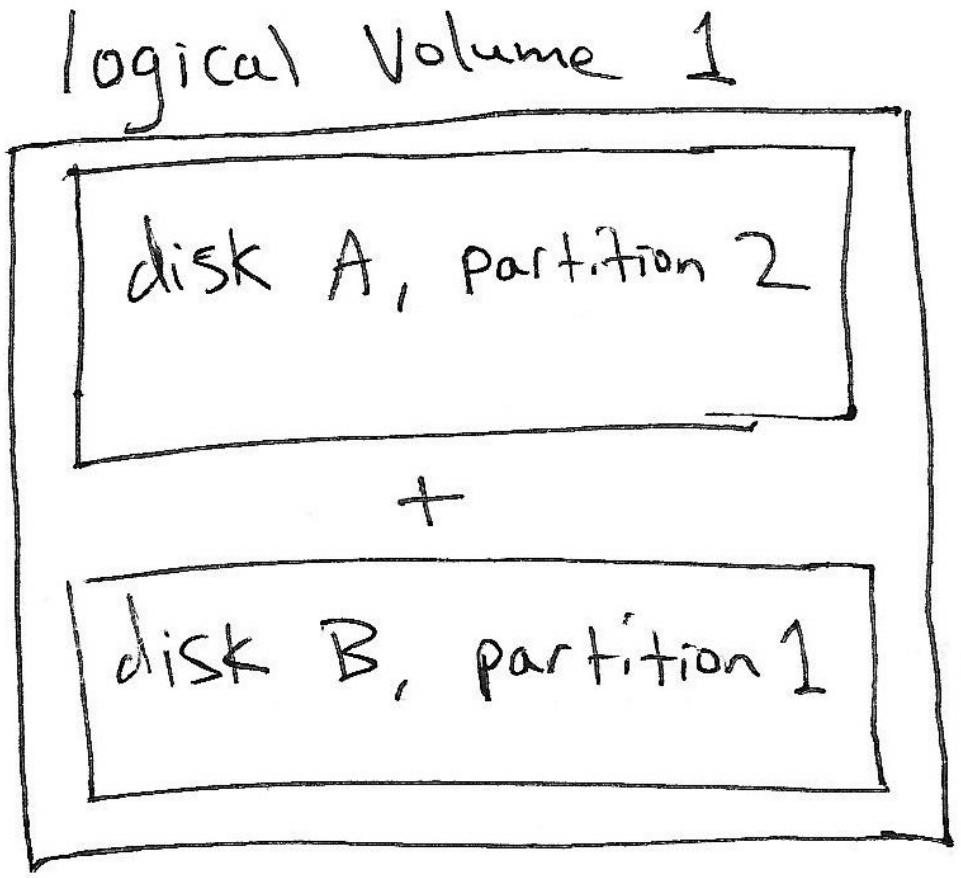
(not drawn to scale)

- Most computers have one physical disk,
- But they may require multiple filesystems.
- A disk partition is a contiguous chunk of the disk that can be formatted to store a filesystem.

At left, we have:

- Three different Linux partitions: /boot, swap, /
- A Windows partition.
  - Each of the partitions may be formatted differently.
- At bootup, initial boot code will present user with a menu to choose Windows or Linux boot.

Logical Volume Management (LVM)



- It's sometimes convenient to combine multiple disk partitions into a bigger **logical volume**.
- The concept is similar to software RAID, but it does not provide performance or redundancy benefits.
- Allows user to increase the size of a filesystem by later adding another disk.
- I think this feature is overused as a default setting on modern Linux distributions.

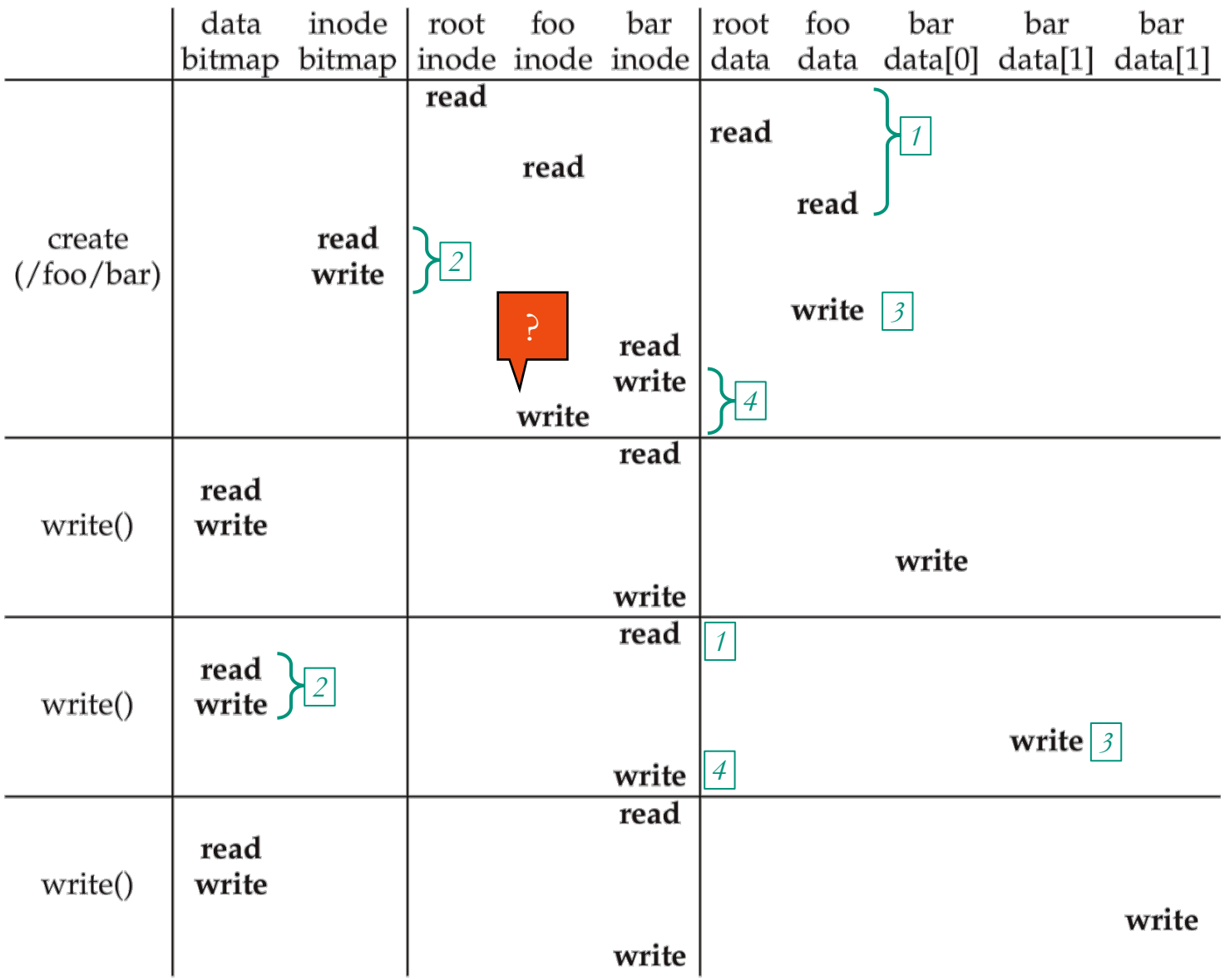
A trace through the filesystem – *open & read*

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[1]
open("/foo/bar")			**read**	**read**	**read**	**read**	**read**			
read()			**read** **write**					**read**		
read()			**read** **write**						**read**	
read()			**read** **write**							**read**

time ↓

- Note that the book's vsfs uses an ***inode bitmap*** to find free inodes.
  - xv6 filesystem simply scans through inodes to find an unused one.

Create & write a file



Create:

1. First, read the parent directory to ensure that name is not already used.
2. Find & claim a free inode.
3. Add <"bar", inode#> to parent directory.
4. Fill-in file metadata.

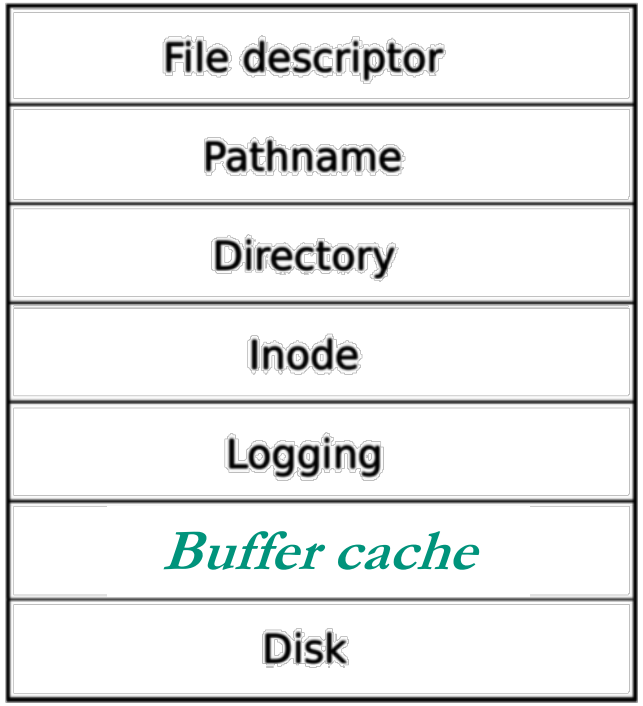
Write:

1. Look for remaining space in existing blocks first.
2. Find & claim a new data block.
3. Write data to new block
4. Point to it in inode

I/O overload

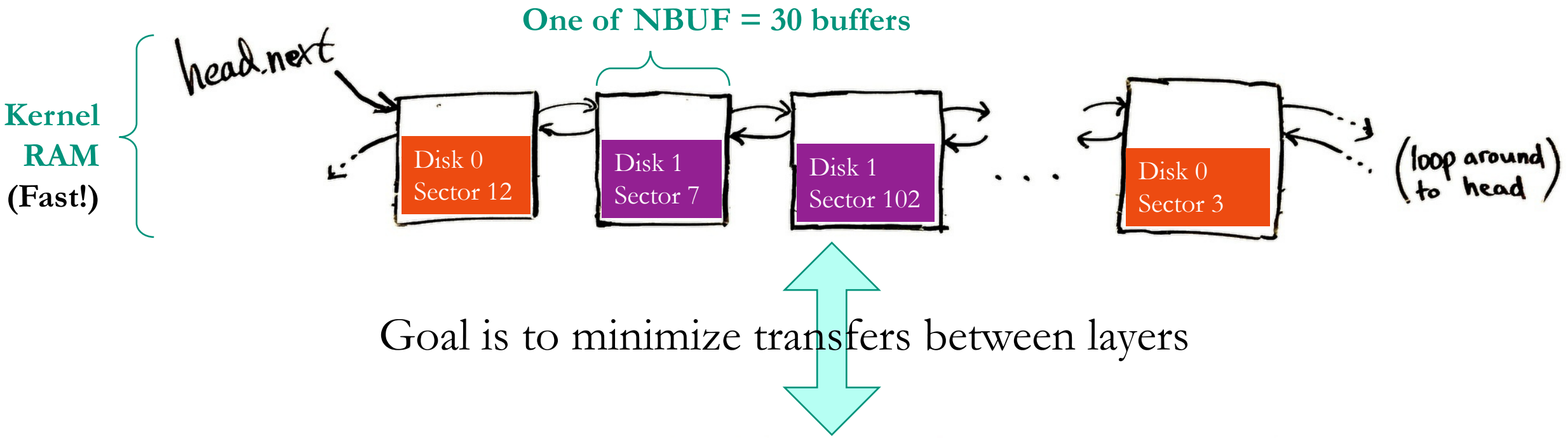
- Filesystem is a complex data structure, requiring reads/writes to many locations of the disk to perform even basic operations:
  - ***Open***: accessed **five** different blocks
  - ***Read***: required **three** I/Os, including a write (to update last accessed time).
  - ***Create***: required **nine** I/Os.

- How to make this fast??
  - Add a ***buffer cache*** layer to the OS storage system.
    - Try to cache disk blocks in RAM whenever possible.
  - Layered design breaks the overall storage system's complexity into a few simple layers.



xv6 buffer cache

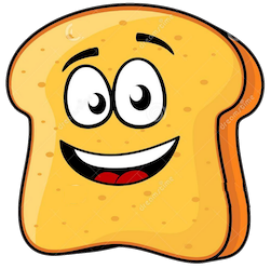
- Upper layers (filesystem) work with buffers, not directly with disk



Buffer Cache goals

1. Cache popular blocks so the disk can be accessed less frequently.
   - Recall that disk has 10,000× greater delay than RAM.
   - ***Reads*** are faster if the disk block is already in memory from a recent access.
   - ***Writes*** can be aggregated.
     - If a thread writes three times briefly to the same file, these can likely be reduced to one write to disk if the writes are delayed.
     - If a thread creates a new file and quickly deletes it, these writes can be skipped altogether.
   - Eventually, changes must be flushed to disk, but there is no rush.
2. Must be careful to prevent two threads from accessing different unsynchronized copies of the disk block.
   - Ie., make the cache **coherent** and avoid race conditions

Buffer cache interface (kernel/bio.c)



Higher storage layers do not access disk directly, instead they call:

- `struct buf* bread(int device, int sector)`
  - Returns a cache buffer with the data at a given location on a given disk.
  - If the sector is already in a buffer, *great!* We avoided a disk read.
  - If another thread using that buffer already, then sleep until it's available.
  - Buffer is **locked** for thread's exclusive use.
- `void bwrite(struct buf* b)`
  - Write buffer's new contents to disk.
  - Must always call bread before bwrite.
- `void brelse(struct buf* b)`
  - **Release the lock** on the buffer (and wake any waiting thread)
  - Move buffer to the head of the queue (MRU)

Example of buffer cache layer in action (kernel/fs.c)

- `write` *syscall* → `sys_write` → `filewrite` → `writei`

```
438     for(tot=0; tot<n; tot+=m, off+=m, src+=m){
439       bp = bread(ip->dev, bmap(ip, off/BSIZE));
440       m = min(n - tot, BSIZE - off%BSIZE);
441       memmove(bp->data + off%BSIZE, src, m);
442       bwrite(bp);
443       brelse(bp);
444     }
```

➢ *Bread* to get buffer & lock

➢ Copy data to the buffer

➢ Flush data

➢ Release lock

Buffer cache details

```
3   // IO Buffer
4   struct buf {
5     int flags;
6     uint dev;
7     uint sector;
8     struct buf *prev; // LRU cache list
9     struct buf *next;
10    struct buf *qnext; // disk queue
11    uchar data[512];
12  };
13  #define B_BUSY  0x1  // buffer is locked by some process
14  #define B_VALID 0x2  // buffer has been read from disk
15  #define B_DIRTY 0x4  // buffer needs to be written to disk
```

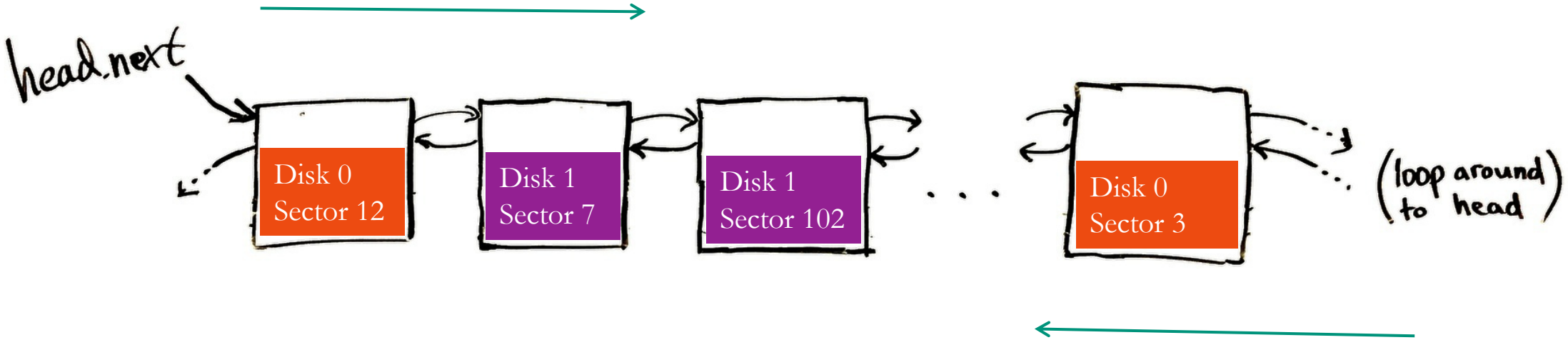
Busy/Locked? Valid? Dirty?

Doubly-linked circular list of buffers

List of buffers waiting to be written to disk.
(Implementing it here is kind of sloppy.)

Most recently used buffer is moved to head by brelse

bread → *bget* looks for a matching buffer starting at head
(disk and sector #s must match)



… and allocates new buffers starting at tail
(buffer must be marked "not busy")

- xv6 panics if a suitable buffer cannot be found.
- No more than 30 (NBUF) concurrent disk accesses are possible!

Unified page cache (Linux)

- Recall that a *unified page cache* allocates physical memory to store both:
  - process' virtual pages, and
  - device block buffers.
- Use a 4 kB page frame to store 1, 2, 4, or 8 buffered disk blocks
  (depending on the filesystem's chosen block size)
- A page fault or a disk I/O can both:
  - evict a resident page or cached disk block.
  - request a physical frame

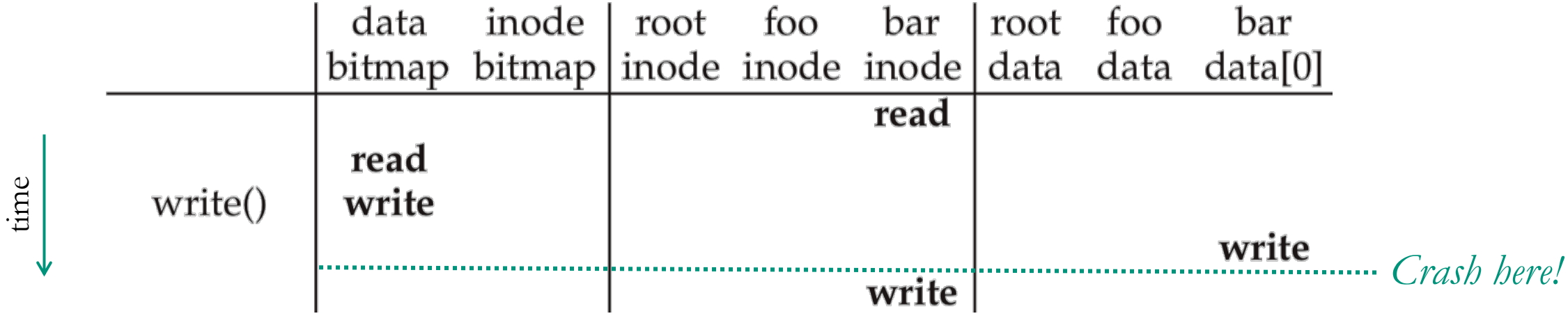
Intermission

"If reëlected, this time I promise not to procrastinate for four years and then try to get all my governing done in one epic all-nighter."

Crash tolerance

- Filesystems are persistent and store important data
- We cannot rely on the user to shut down the system gracefully
  - Power outages happen
  - Kernel may panic
  - USB plug may be yanked out
- But, filesystem is a complex data structure with ***critical sections***.
  - The concern is not race conditions, but **partial commit**.
  - Some filesystem transactions must be performed atomically – "all or none," otherwise the filesystem will not be self-consistent.
- It's OK to interrupt a write, as long as it leaves the file looking like it was partially written, rather than corrupting the file or filesystem.

Crash example (while writing to /foo/bar)

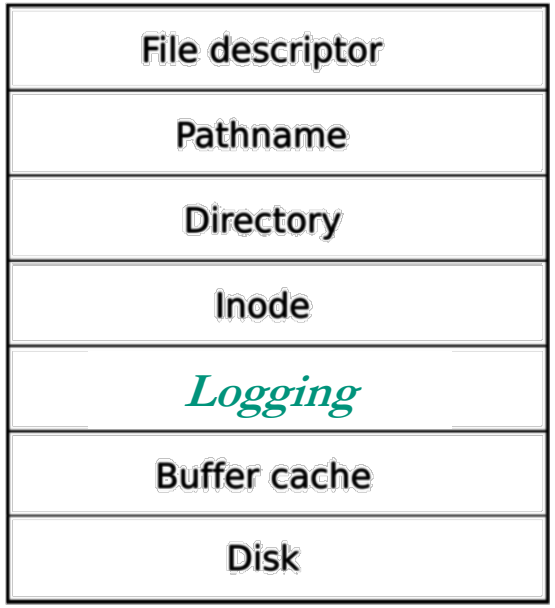


- A crash before the write to the file's inode would ***leak*** a data block
  - The data bitmap was updated to reserve a direct block, but we were not able to finish by making the file's inode refer to that block.
  - This block is **lost forever**. 😢
- Depending on the FS implementation, more serious problems are possible:
  - Inode might refer to a block that has not been written yet, adding garbage data.
  - … or to a block that was freed in the bitmap, making two files share the block.

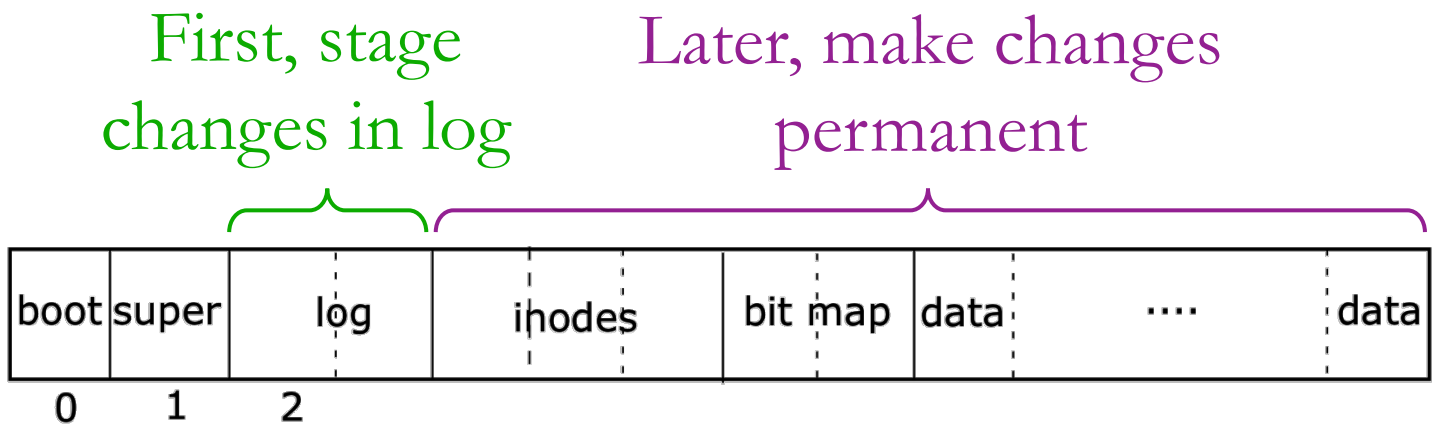
xv6 logging layer makes filesystem *transactional*

- Group related writes into a transaction.
- Call **end_op** to atomically commit the transaction.
- Example usage in a syscall:

```
begin_op();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
end_op();
```



Write-ahead logging: write the transaction to disk ***twice***:



1. Write the blocks to the log, a reserved part of the disk.
   - This makes a durable record of the transaction you plan to commit.
   - Continue putting all writes to the log, until ***commit*** is called.
2. On commit, write a commit message to the log, then start writing all of the logged writes where they belong on disk.
   - Clear the log after everything is written again.

What happens after a crash?

The next time the machine is booted, we deal with it as follows:

1. Crash occurred *before commit*:
   - There is data in the log, but no commit message.
   - Just clear the log to ***roll back*** the transaction.
2. Crash occurred *after commit*, while writing data to main part of disk.
   - We don't know how much of the transaction was finished.
   - However, the log tells us exactly what must be done!
   - ***Replay*** the transaction (from the beginning), then clear the log.
3. If the log is empty, do nothing because there were no outstanding transactions.

Code in more depth

```
begin_op();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
end_op();
```

- **`begin_op`** waits until the logging system is not committing and there is enough free space in the log.
- **`log_write`** reserves a place for the block in the log, but does not write to disk yet.
  - So far, everything is being done in memory, using the buffer cache.
  - This allows multiple log_writes to the same block to be *absorbed*.
- **`end_op`** writes the transaction to disk in the log, then in the destination sectors.

Note that transactions must be small enough to fit in the log.

Alternative crash recovery techniques

- Older filesystems (ext2, FAT32) have no log.
- Instead, a ***scavenger*** program (FSCK) runs on reboot to check for filesystem inconsistencies.
  - This can take hours on a large filesystem.
  - FSCK is only run if the system detects an unclean reboot
- ext3 filesystem added a log (journal) to ext2.

- Note that logging can **double** disk write load.
- However, if you are careful you can limit logging to just the *metadata* (inode writes) and write data once to disk.

Recap: 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***.