

CS162
Operating Systems and
Systems Programming
Lecture 23

Filesystems 3: Buffer Cache, Reliability, Transactions

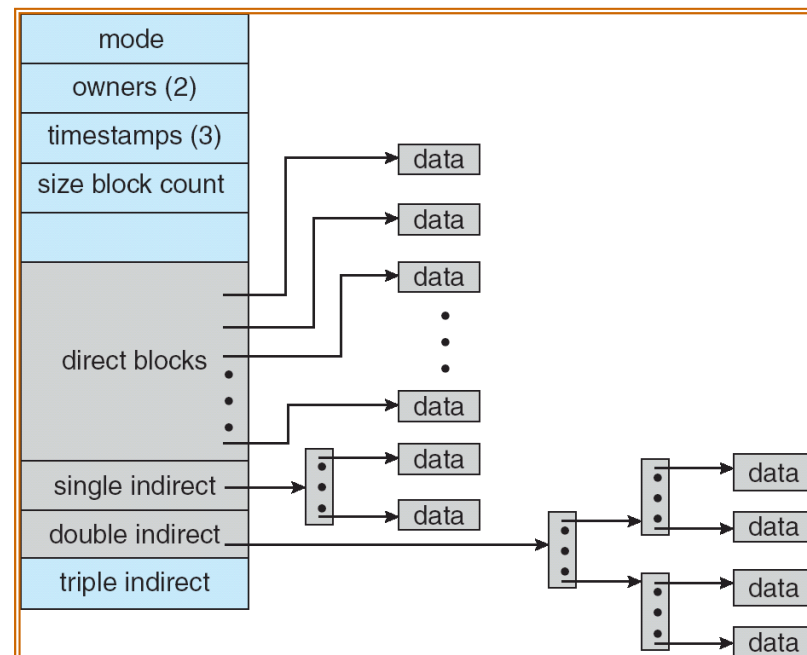
April 21st, 2026

Prof. John Kubiatowicz

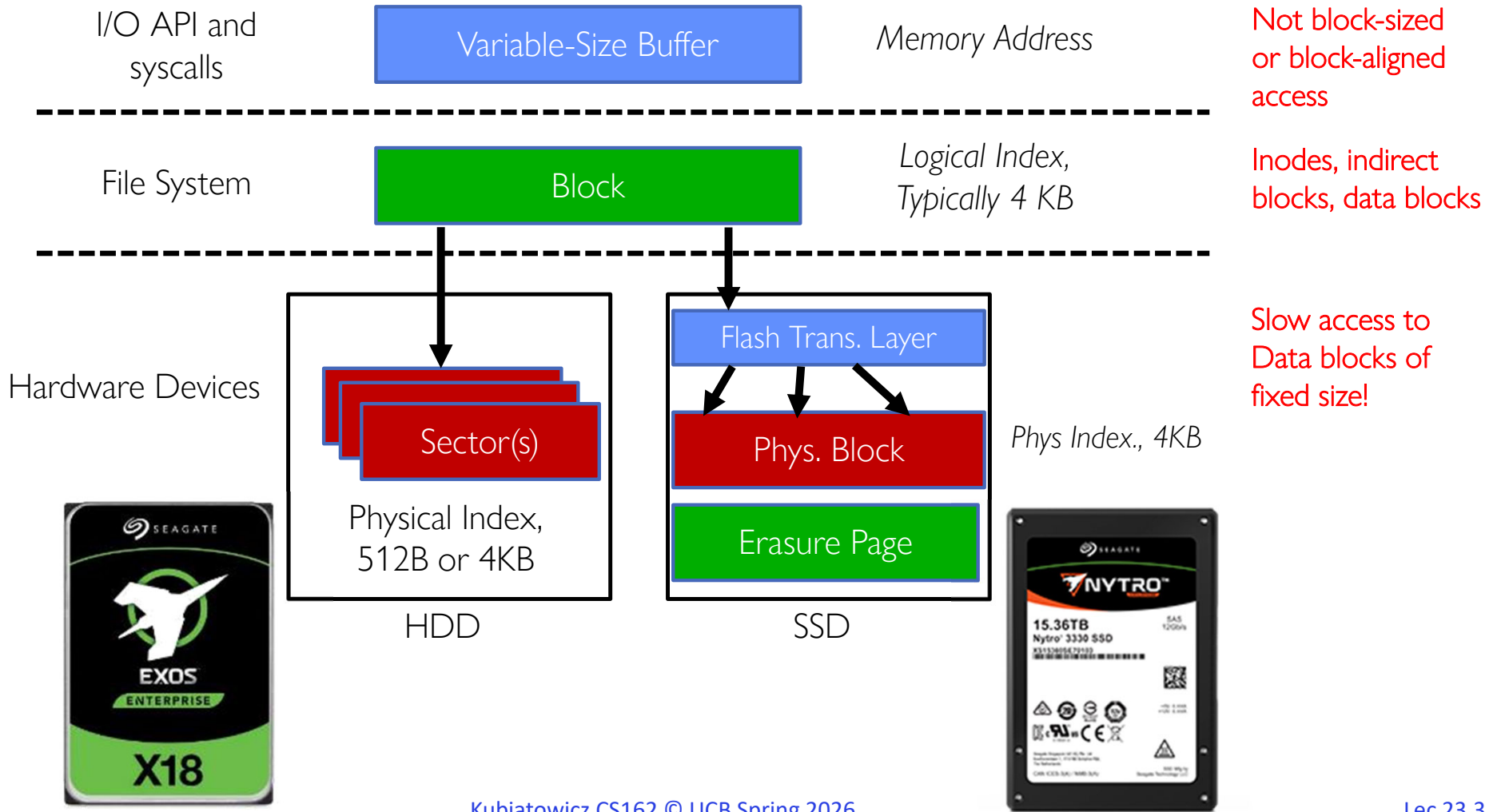
<http://cs162.eecs.Berkeley.edu>

Recall: Original Multilevel Indexed Files (4.1 BSD, 1981)

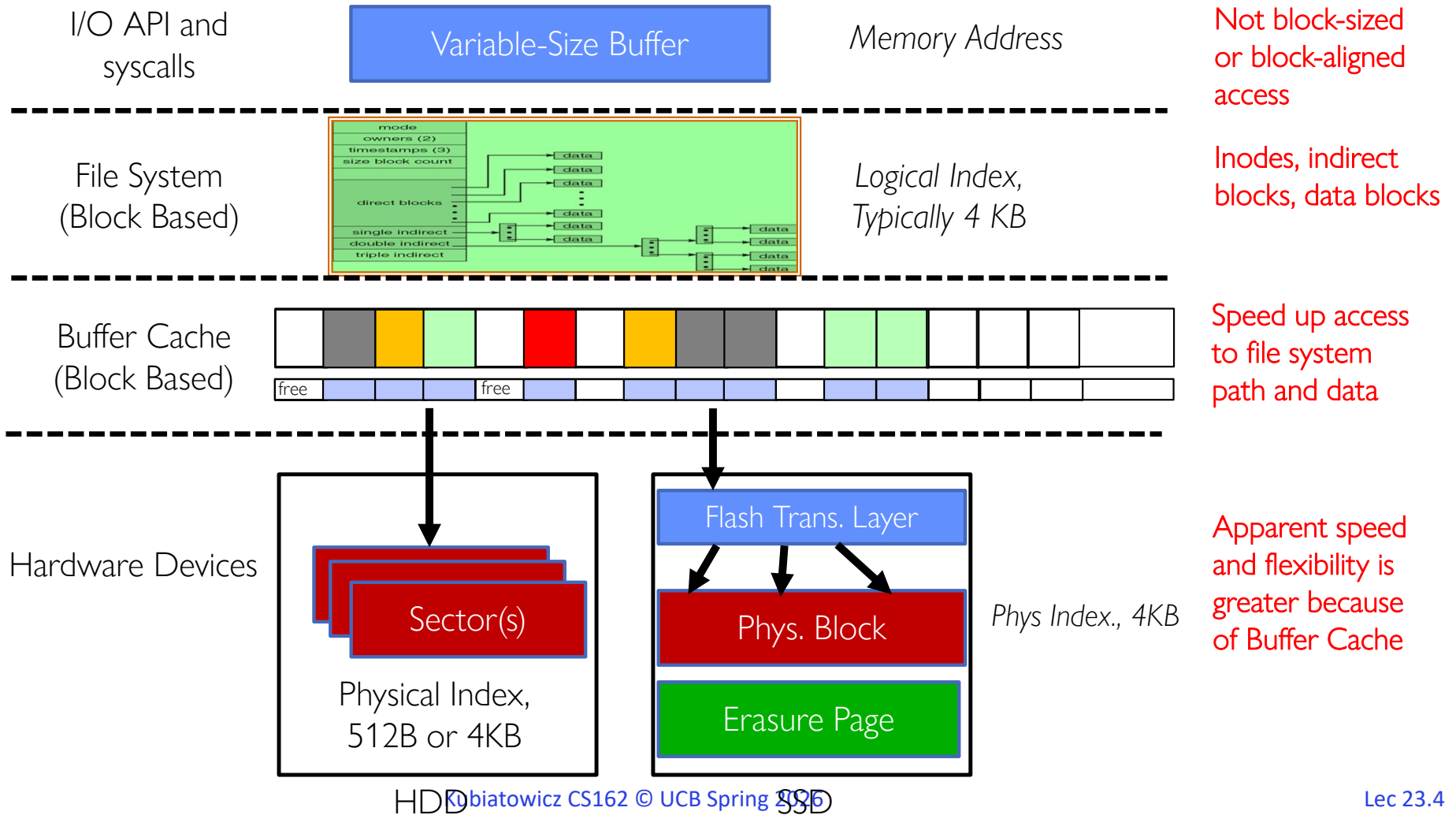
- Original 4.1BSD file system
 - 10 “direct pointers” to blocks (1KB blocks)
 - 1 “single indirect ptr” (to block with 256 ptrs to blocks)
 - 1 “double indirect ptr” (to block with 256 indirect ptrs)
 - Metadata associated with FILE, not directory entry (like FAT)
- Sample file in multilevel indexed format:
 - How many accesses for block #23?
(assume file header accessed on open)?
 - » Two: One for indirect block, one for data
 - How about block #5?
 - » One: One for data
 - Block #340?
 - » Three: double indirect block, indirect block, and data
- UNIX 4.1 Pros and cons
 - Pros: Simple (more or less)
Files can easily expand (up to a point)
Small files particularly cheap and easy, large files can be handled!
 - Cons: Lots of seeks
Very large files must read many indirect block (four I/Os per block!)



Recall: From Storage to File Systems



Need for Cache Between FileSystem and Devices



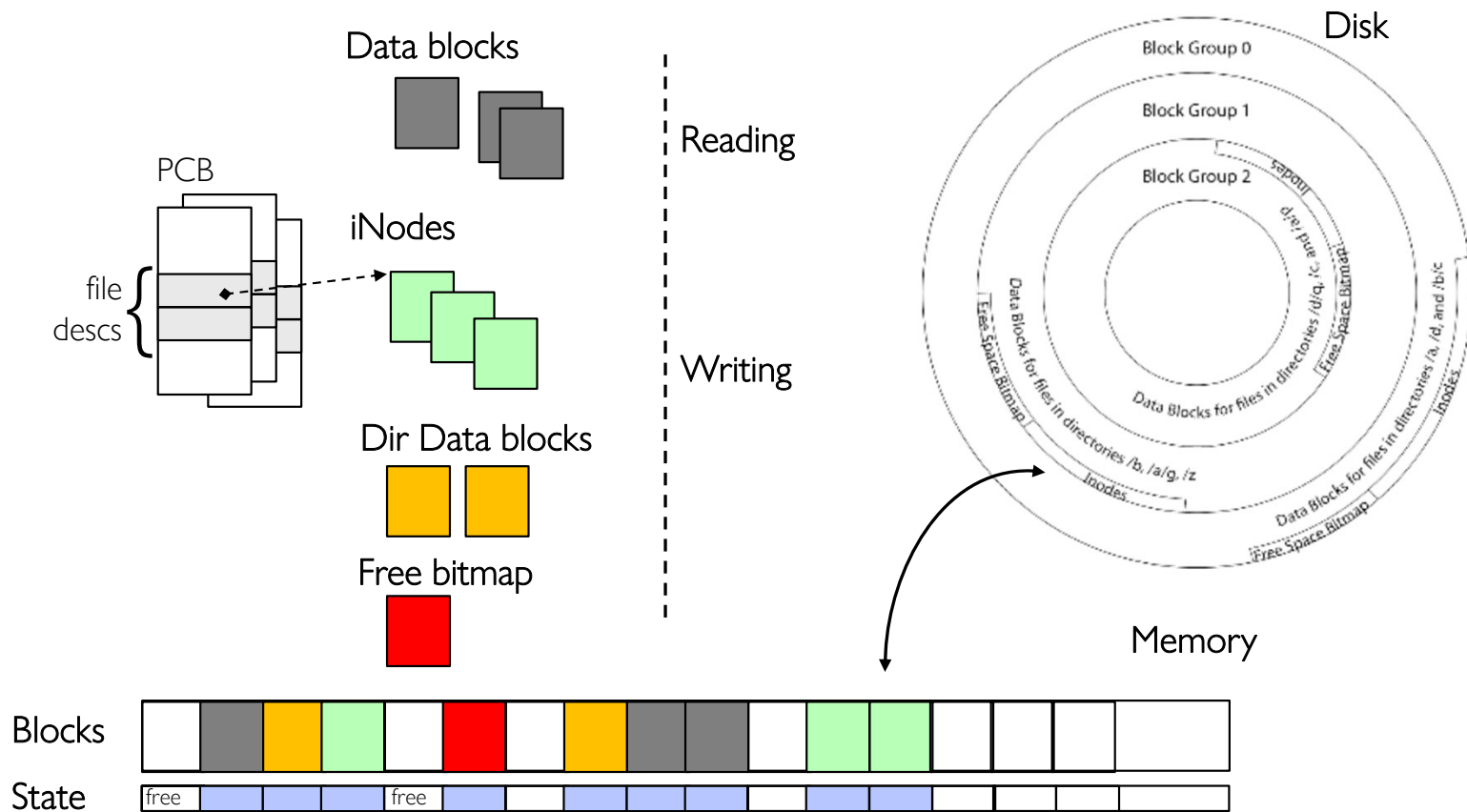
Buffer Cache: Motivation



- Kernel *must* copy disk blocks to memory (somewhere) to access their contents and write them back if modified
 - Could be data blocks, inodes, directory contents, etc.
 - Possibly dirty (modified and not yet written back)
- Key Idea: Exploit locality by caching disk data in memory
 - Name translations: Mapping from paths→inodes
 - Disk blocks: Mapping from block address→disk content
- **Buffer Cache:** Memory used to cache kernel resources, including disk blocks and name translations
 - Can contain “dirty” blocks (with modifications not on disk)

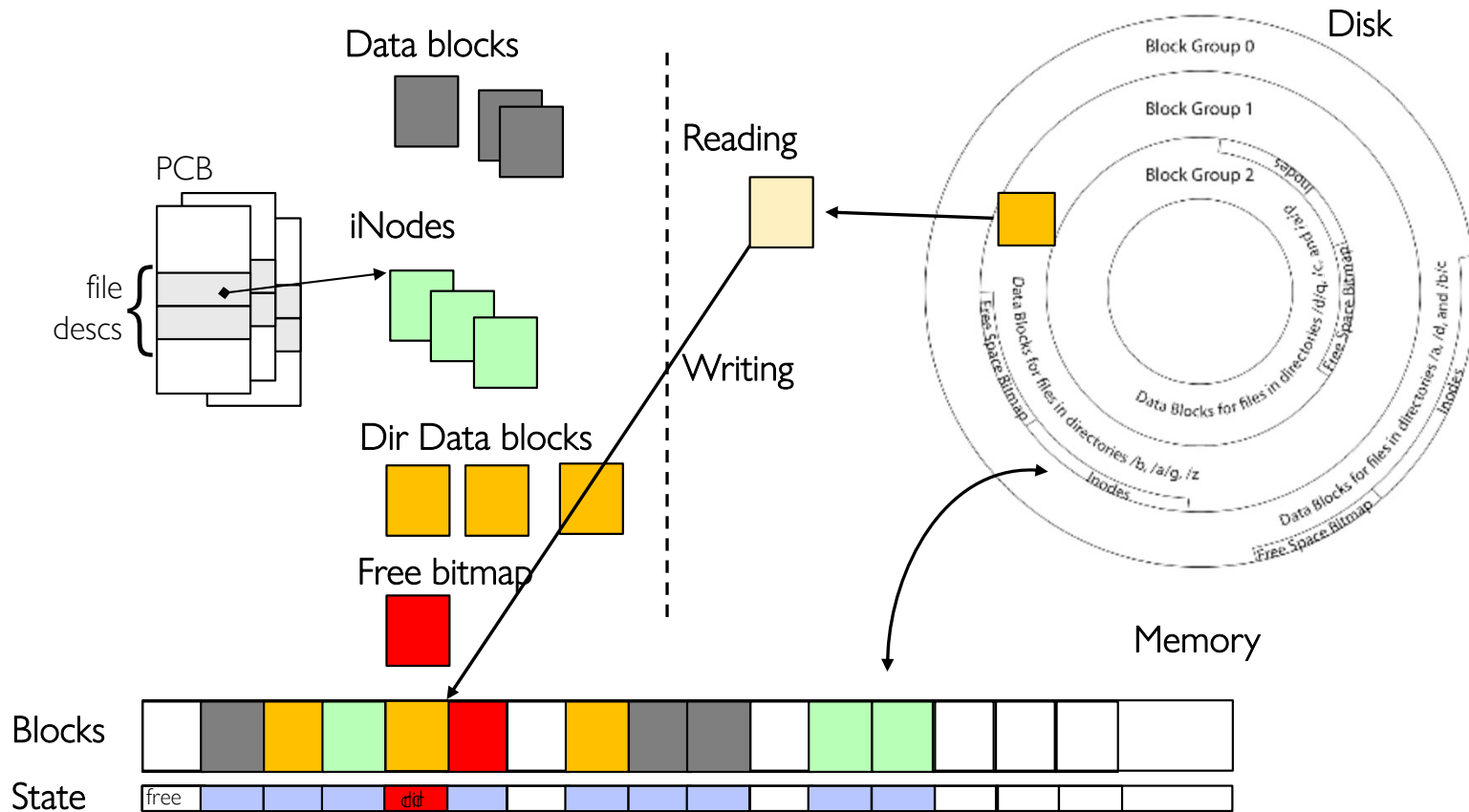
File System Buffer Cache

- OS implements a cache of disk blocks for efficient access to data, directories, inodes, freemap



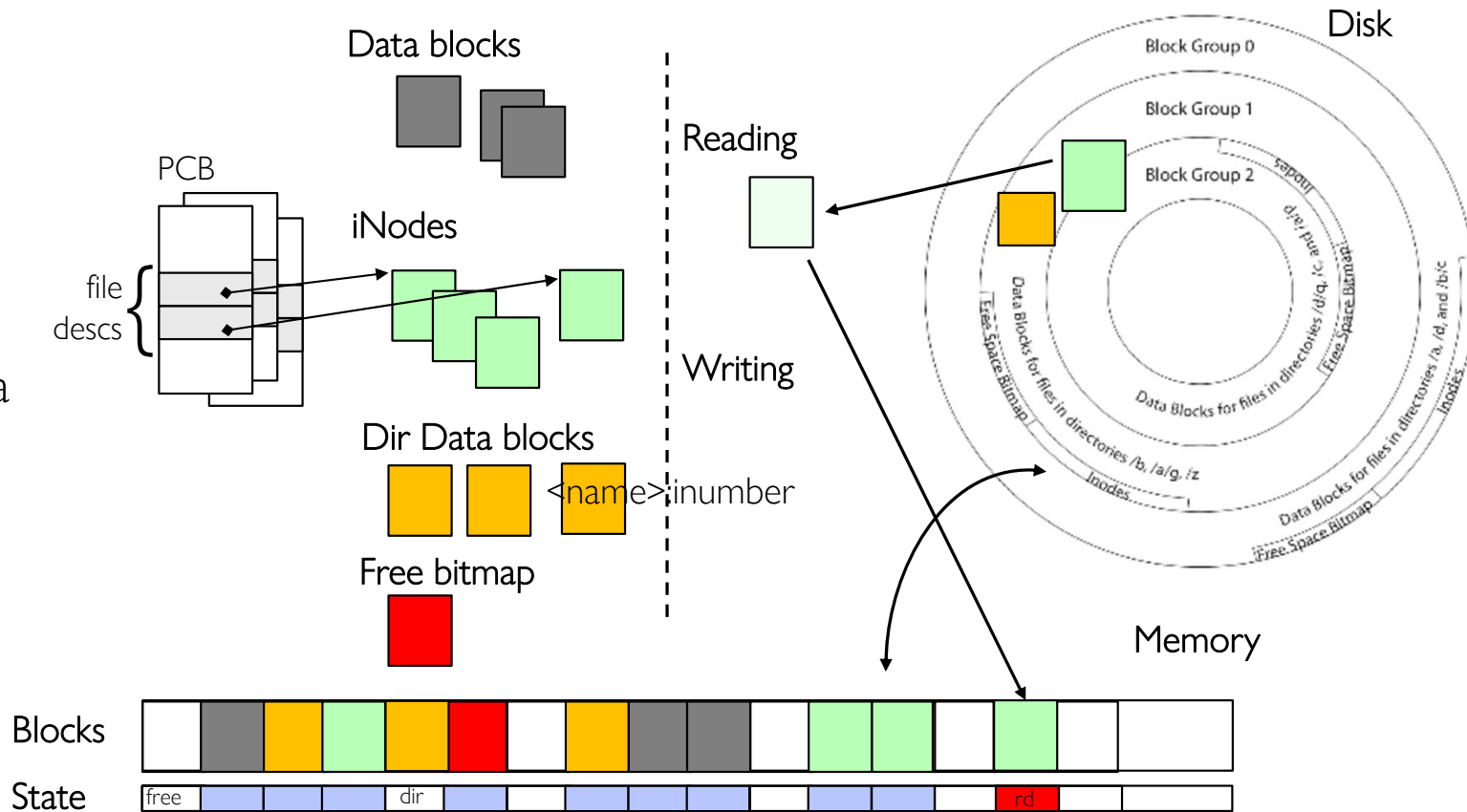
File System Buffer Cache: open

- Directory lookup repeat as needed:
 - load block of directory
 - search for map



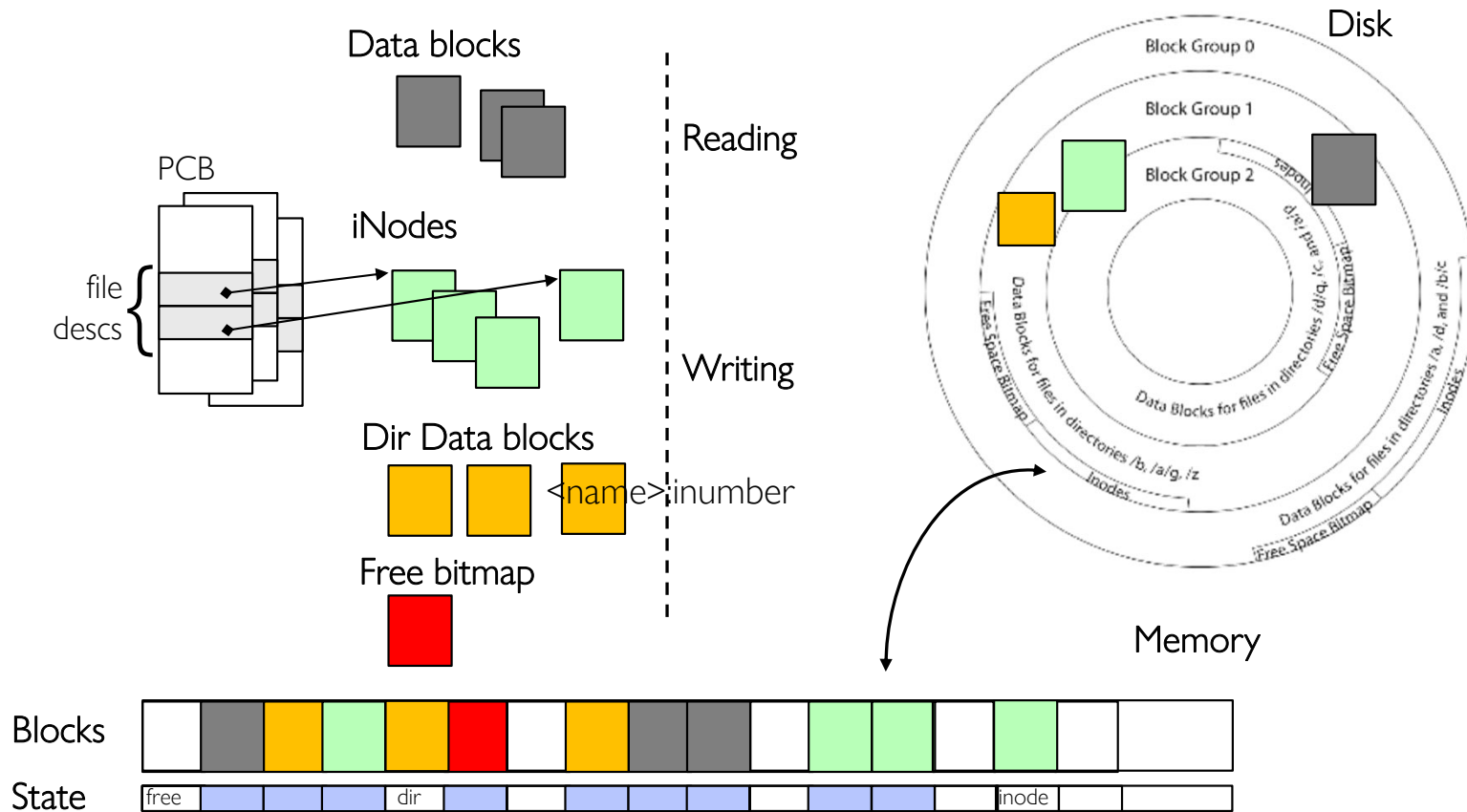
File System Buffer Cache: open

- Directory lookup repeat as needed:
 - load block of directory
 - search for map
- Create reference via open file descriptor



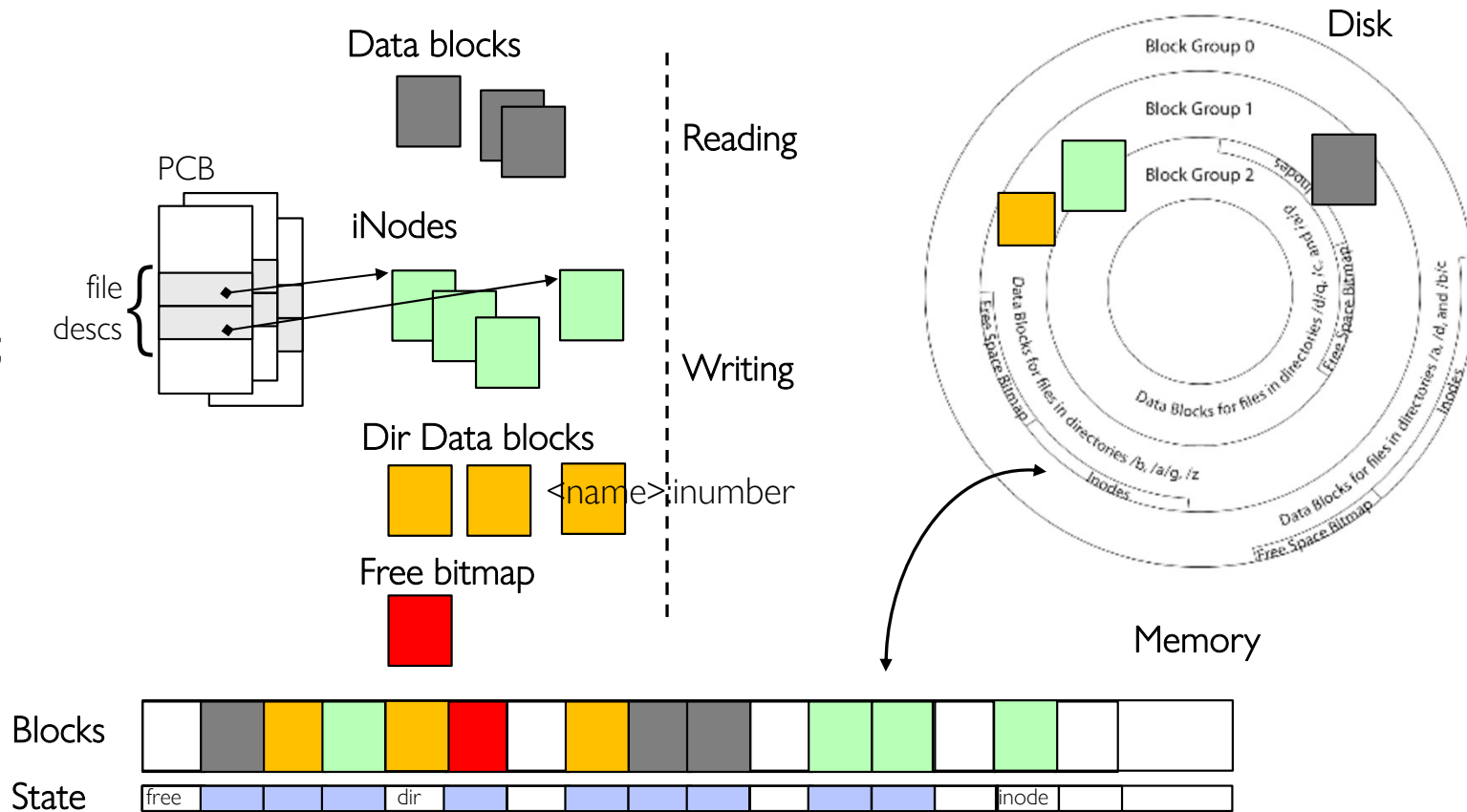
File System Buffer Cache: Read?

- Read Process
 - From inode, traverse index structure to find data block
 - load data block
 - copy all or part to read data buffer



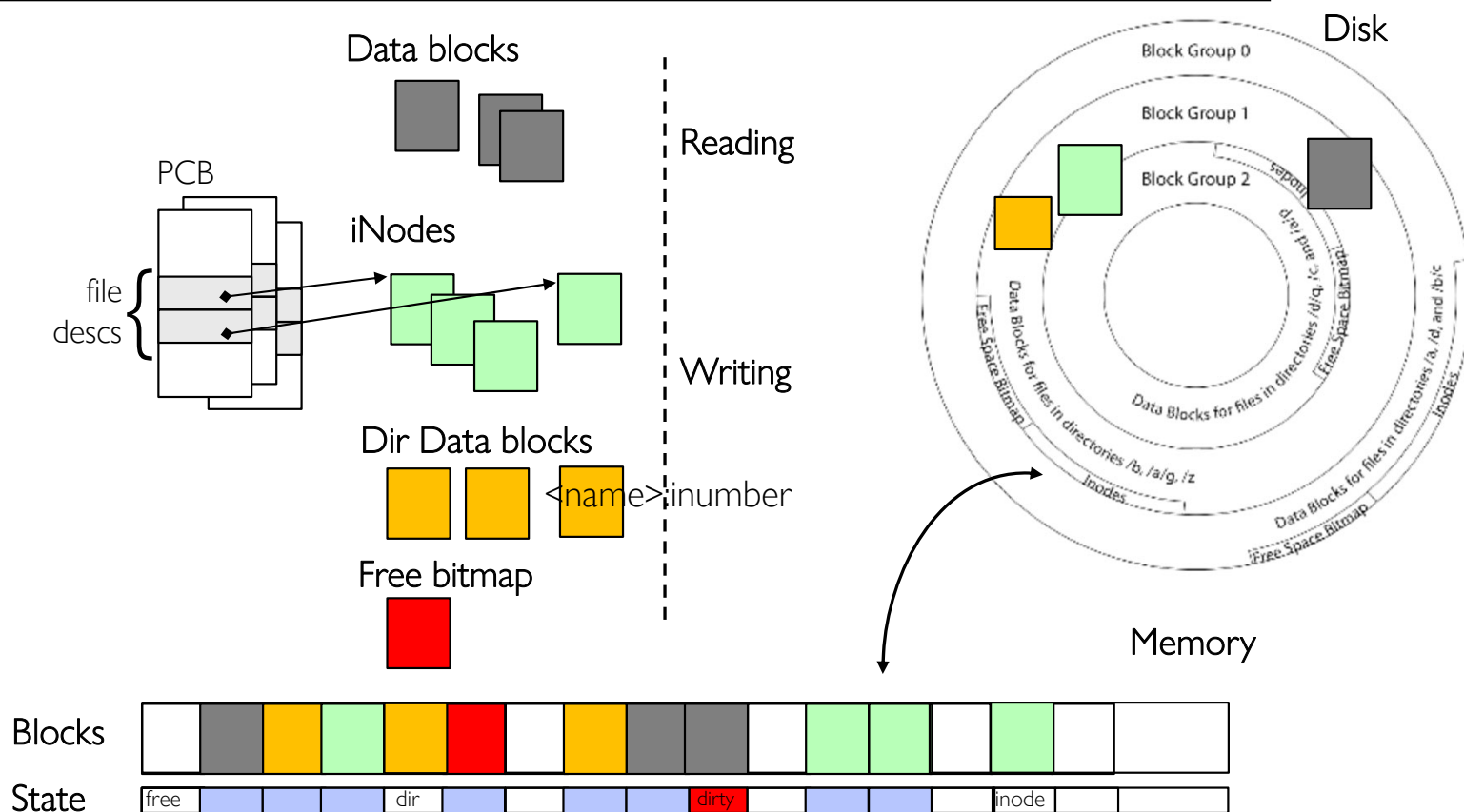
File System Buffer Cache: Write?

- Process similar to read, but may allocate new blocks (update free map), blocks need to be written back to disk; inode?



File System Buffer Cache: Eviction?

- Blocks being written back to disc go through a transient state



Buffer Cache Discussion

- Implemented entirely in OS software
 - Unlike memory caches and TLB
- Blocks go through transitional states between free and in-use
 - Being read from disk, being written to disk
 - Other processes can run, etc.
- Blocks are used for a variety of purposes
 - inodes, data for directories and files, freemap
 - OS maintains pointers into them
- Termination – e.g., process exit – open, read, write
 - Process buffers (i.e. from fopen, fwrite, fread) typically written out on exit
 - Buffer cache buffers not necessarily pushed to disk immediately
- Replacement – what to do when it fills up?

File System Caching

- Replacement policy? LRU
 - Can afford overhead full LRU implementation
 - Advantages:
 - » Works very well for name translation
 - » Works well in general as long as memory is big enough to accommodate a host's working set of files.
 - Disadvantages:
 - » Fails when some application scans through file system, thereby flushing the cache with data used only once
 - » Example: `find . -exec grep foo {} \;`
- Other Replacement Policies?
 - Some systems allow applications to request other policies
 - Example, 'Use Once':
 - » File system can discard blocks as soon as they are used

File System Caching (con't)

- Cache Size: How much memory should the OS allocate to the buffer cache vs virtual memory?
 - Too much memory to the file system cache \Rightarrow won't be able to run many applications
 - Too little memory to file system cache \Rightarrow many applications may run slowly (disk caching not effective)
 - Solution: adjust boundary dynamically so that the disk access rates for paging and file access are balanced

File System Prefetching

- **Read Ahead Prefetching:** fetch sequential blocks early
 - Key Idea: exploit fact that most common file access is sequential by prefetching subsequent disk blocks ahead of current read request
 - Elevator algorithm can efficiently interleave prefetches from concurrent applications
- How much to prefetch?
 - Too much prefetching imposes delays on requests by other applications
 - Too little prefetching causes many seeks (and rotational delays) among concurrent file requests

Delayed Writes

- Buffer cache is a writeback cache (writes are termed “Delayed Writes”)
- **write()** copies data from user space to kernel buffer cache
 - Quick return to user space
- **read()** is fulfilled by the cache, so **reads** see the results of **writes**
 - Even if the data has not reached disk
- When does data from a **write** syscall finally reach disk?
 - When the buffer cache is full (e.g., we need to evict something)
 - When the buffer cache is flushed periodically (in case we crash)

Delayed Writes (Advantages)

- Performance advantage: return to user quickly without writing to disk!
- Disk scheduler can efficiently order lots of requests
 - Elevator Algorithm can rearrange writes to avoid random seeks
- Delay block allocation:
 - May be able to allocate multiple blocks at same time for file, keep them contiguous
- Some files never actually make it all the way to disk
 - Many short-lived files!

Buffer Caching vs. Demand Paging

- Replacement Policy?
 - Demand Paging: LRU is infeasible; use approximation (like NRU/Clock)
 - Buffer Cache: LRU is OK
- Eviction Policy?
 - Demand Paging: evict not-recently-used pages when memory is close to full
 - Buffer Cache: write back dirty blocks periodically, even if used recently
 - » Why? To minimize data loss in case of a crash

Dealing with Persistent State

- Buffer Cache: write back dirty blocks periodically, even if used recently
 - Why? To minimize data loss in case of a crash
 - Linux does periodic flush every 30 seconds
- Not foolproof! Can still crash with dirty blocks in the cache
 - What if the dirty block was for a directory?
 - » Lose pointer to file's inode (leak space)
 - » File system now in inconsistent state ☹️

Takeaway: File systems need recovery mechanisms

Administrative

- Midterm 3 on Thursday, April 30
 - All topics up to previous Tuesday (4/28) are in scope
 - Closed book, 3 pages, double-sided *handwritten* notes.
- Project 3 is out!
 - Get moving, since the end of term is approaching rapidly!
 - (That light you see is an approaching train!)
 - Design document due date moved to 4/18 😊
- Other deadlines:
 - HW5 Due ⇒ Friday 4/24
 - HW6 Release ⇒ Saturday 4/25
 - HW6 Due ⇒ Thursday 5/7

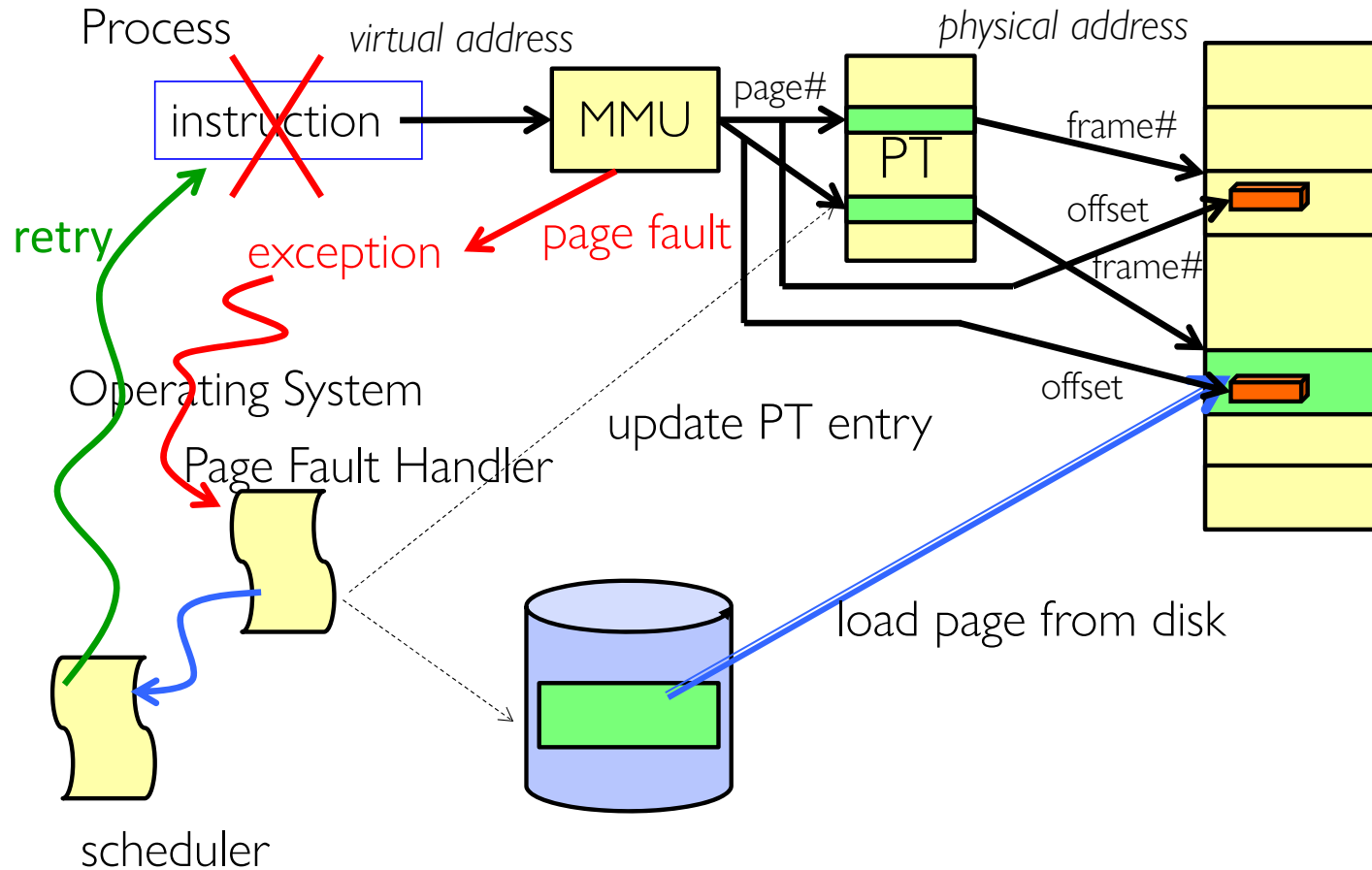
Class Attendance:
4/21/2026



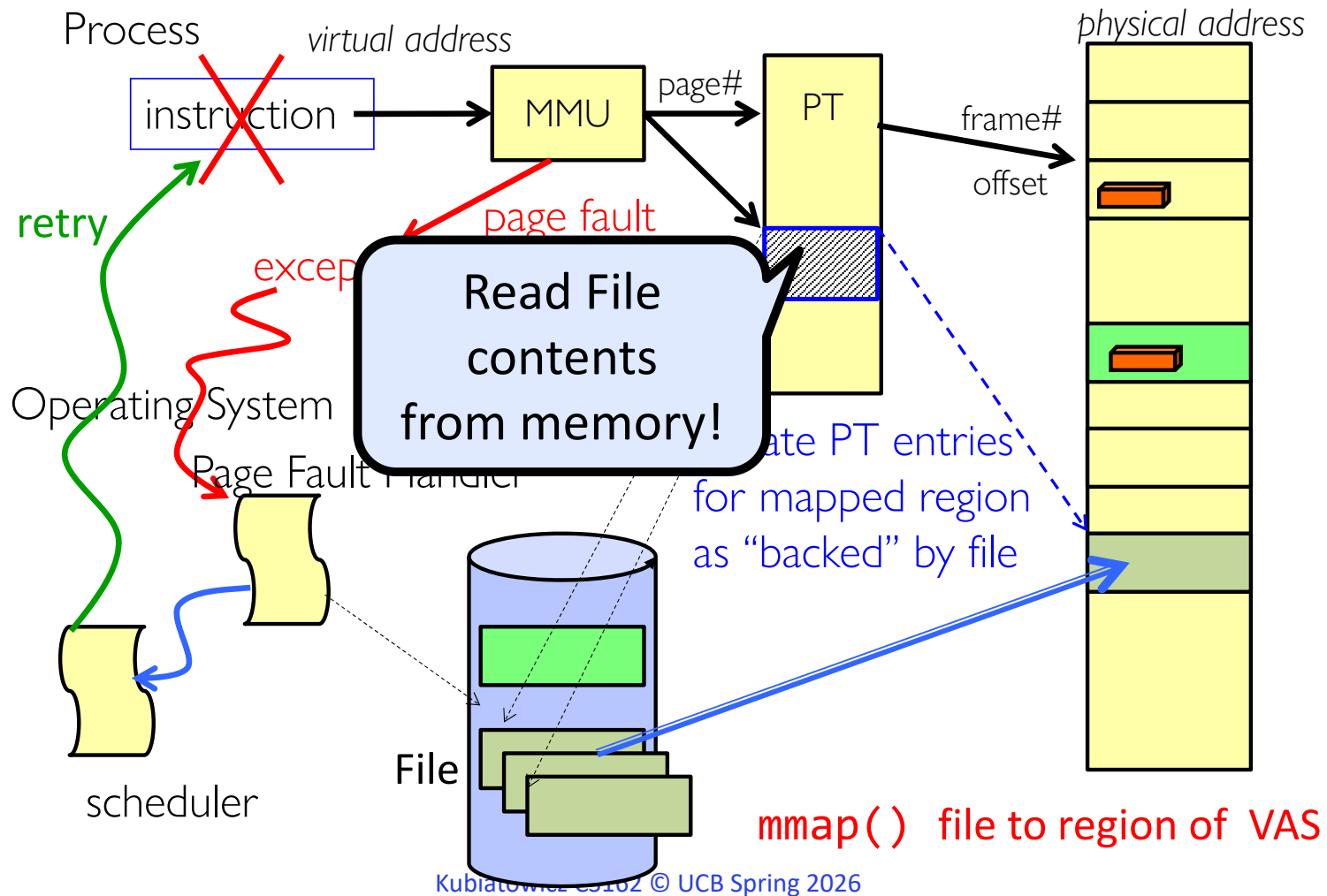
Memory Mapped Files

- Traditional I/O involves explicit transfers between buffers in process address space to/from regions of a file
 - This involves multiple copies into caches in memory, plus system calls
- What if we could “map” the file directly into an empty region of our address space
 - Implicitly “page it in” when we read it
 - Write it and “eventually” page it out
- Data in Buffer Cache already!
- Executable files are treated this way when we `exec` the process!!

Recall: Who Does What, When?



Using Paging to mmap () Files



mmap () system call

MMAP(2)	BSD System Calls Manual	MMAP(2)
NAME <code>mmap</code> -- allocate memory, or map files or devices into memory		
LIBRARY Standard C Library (libc, -lc)		
SYNOPSIS <code>#include <sys/mman.h></code> <code>void *</code> <code>mmap(void *addr, size_t len, int prot, int flags, int fd,</code> <code>off_t offset);</code>		
DESCRIPTION The <code>mmap()</code> system call causes the pages starting at <code>addr</code> and continuing for at most <code>len</code> bytes to be mapped from the object described by <code>fd</code> , starting at byte offset <code>offset</code> . If <code>offset</code> or <code>len</code> is not a multiple of the page size, the mapped region may extend past the specified range.		

- May map a specific region or let the system find one for you
 - Tricky to know where the holes are
- Used both for manipulating files and for sharing between processes

An mmap() Example

```
#include <sys/mman.h> /* also stdio.h, stdlib.h, string.h,fcntl.h,unistd.h */
int something = 162;

int main (int argc, char *argv[]) {
    int myfd;
    char *mfile;

    printf("Data at: %16lx\n", (long) something);
    printf("Heap at : %16lx\n", (long) something);
    printf("Stack at: %16lx\n", (long) something);

    /* Open the file */
    myfd = open(argv[1], O_RDWR | O_CREAT, 0666);
    if (myfd < 0) { perror("open failed!"); return(-1); }

    /* map the file */
    mfile = mmap(0, 10000, PROT_READ|PROT_WRITE, MAP_SHARED, myfd, 0);
    if (mfile == MAP_FAILED) {perror("mmap failed!"); return(-1); }

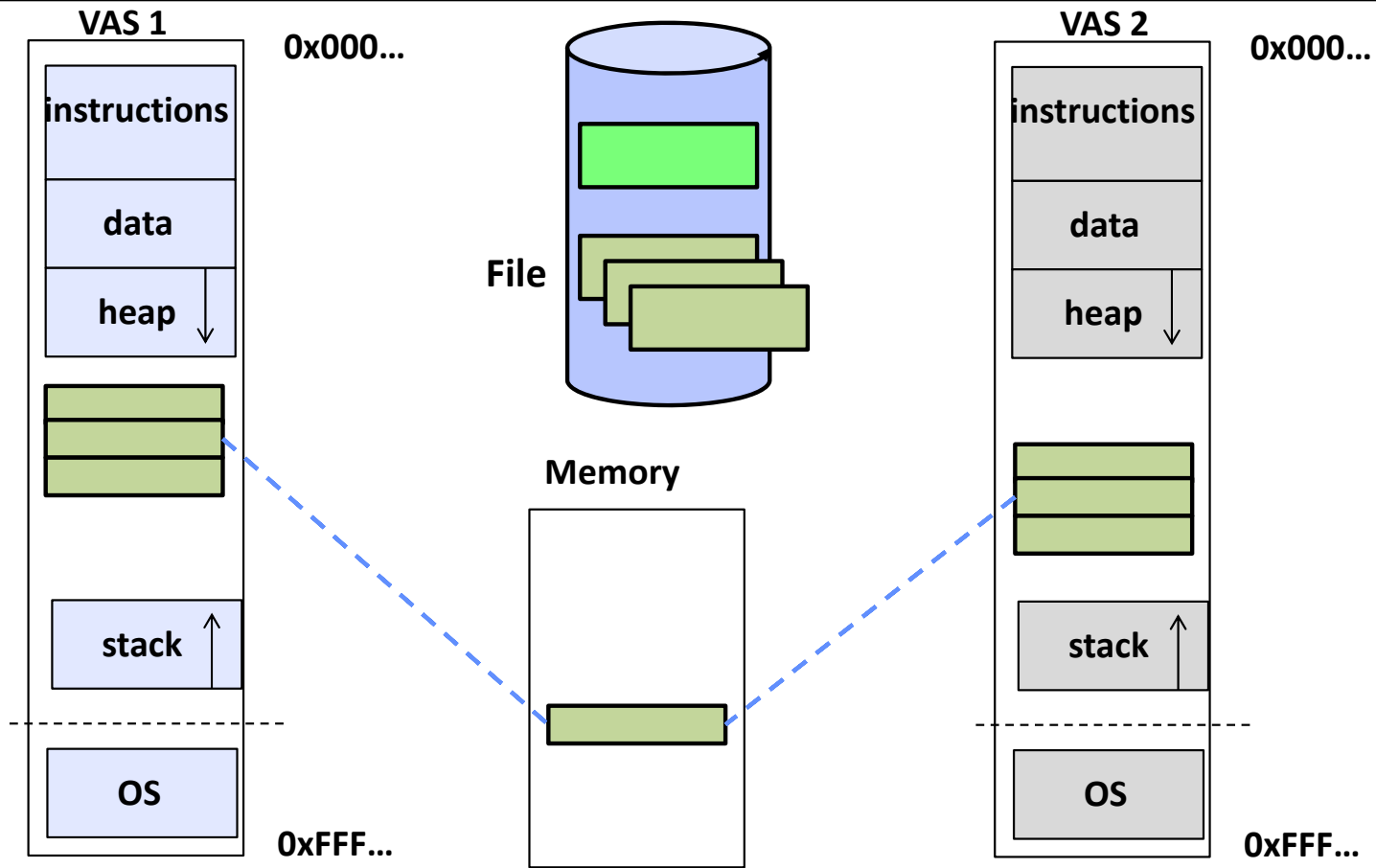
    printf("mmap at : %16lx\n", (long) something);

    puts(mfile);
    strcpy(mfile+20,"Let's write over");
    close(myfd);
    return 0;
}
```

```
$ ./mmap test
Data at:          105d63058
Heap at :         7f8a33c04b70
Stack at:         7fff59e9db10
mmap at :         105d97000
This is line one
This is line two
This is line three
This is line four
```

```
$ cat test
This is line one
This is line two
Let's write over its line three
This is line four
```

Sharing through Mapped Files



- Also: anonymous memory between parents and children
 - no file backing – just swap space

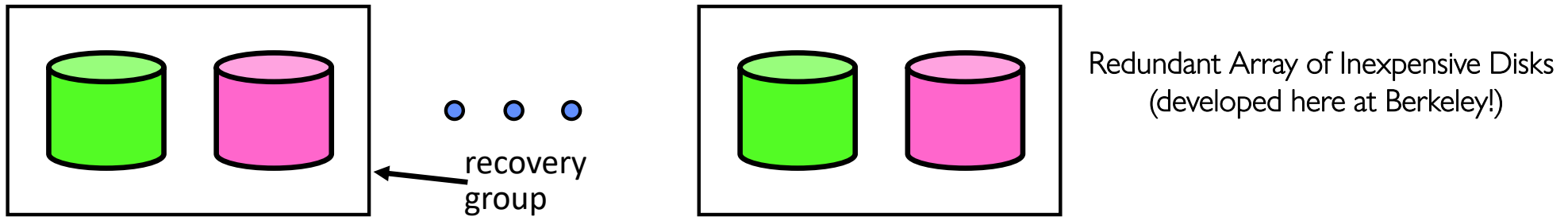
Important “ilities”

- **Availability:** the probability that the system can accept and process requests
 - Measured in “nines” of probability: e.g. 99.9% probability is “3-nines of availability”
 - Key idea here is independence of failures
- **Durability:** the ability of a system to recover data despite faults
 - This idea is fault tolerance applied to data
 - Doesn’t necessarily imply availability: information on pyramids was very durable, but could not be accessed until discovery of Rosetta Stone
- **Reliability:** the ability of a system or component to perform its required functions under stated conditions for a specified period of time (IEEE definition)
 - Usually stronger than simply availability: means that the system is not only “up”, but also working correctly
 - Includes availability, security, fault tolerance/durability
 - Must make sure data survives system crashes, disk crashes, other problems

How to Make File Systems more Durable?

- Disk blocks contain Reed-Solomon error correcting codes (ECC) to deal with small defects in disk drive
 - Can allow recovery of data from small media defects
- Make sure writes survive in short term
 - Either abandon delayed writes or
 - Use special, battery-backed RAM (called non-volatile RAM or **NVRAM**) for dirty blocks in buffer cache
- Make sure that data survives in long term
 - Need to **replicate!** More than one copy of data!
 - Important element: **independence of failure**
 - » Could put copies on one disk, but if disk head fails...
 - » Could put copies on different disks, but if server fails...
 - » Could put copies on different servers, but if building is struck by lightning....
 - » Could put copies on servers in different continents...

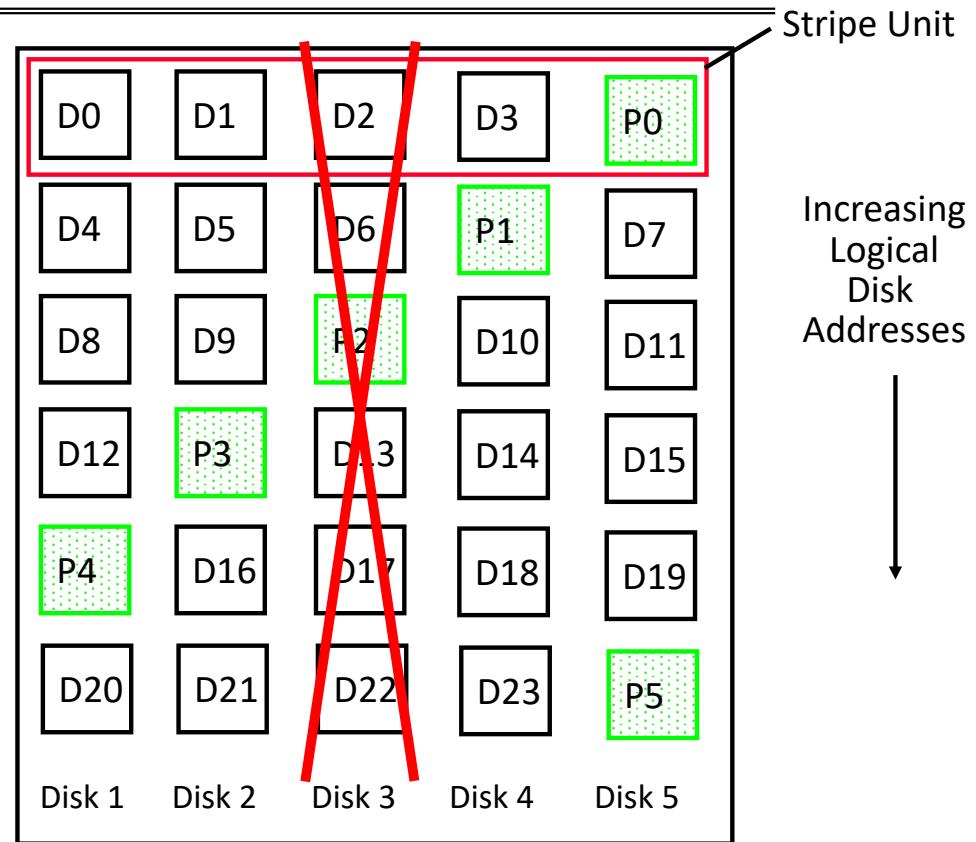
RAID 1: Disk Mirroring/Shadowing



- Each disk is fully duplicated onto its “shadow”
 - For high I/O rate, high availability environments
 - Most expensive solution: 100% capacity overhead
- Bandwidth sacrificed on write:
 - Logical write = two physical writes
 - Highest bandwidth when disk heads and rotation synchronized (challenging)
- Reads may be optimized
 - Can have two independent reads to same data
- Recovery:
 - Disk failure \Rightarrow replace disk and copy data to new disk
 - Hot Spare: idle disk attached to system for immediate replacement

RAID 5+: High I/O Rate Parity

- Data striped across multiple disks
 - Successive blocks stored on successive (non-parity) disks
 - Increased bandwidth over single disk
- Parity block (in green) constructed by XORing data blocks in stripe
 - $P_0 = D_0 \oplus D_1 \oplus D_2 \oplus D_3$
 - Can destroy any one disk and still reconstruct data
- Suppose Disk 3 fails, then can reconstruct:
 $D_2 = D_0 \oplus D_1 \oplus D_3 \oplus P_0$



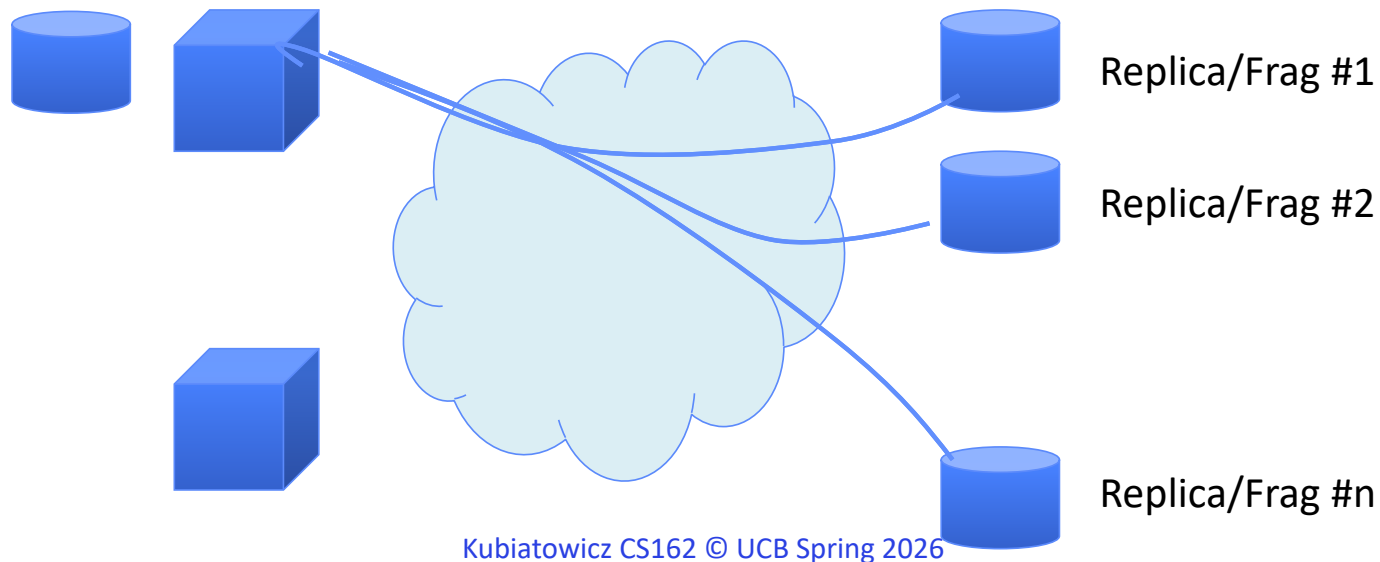
- Can spread information widely across internet for durability
 - RAID algorithms work over geographic scale

RAID 6 and other Erasure Codes

- In general: RAIDX is an “erasure code”
 - Must have ability to know which disks are bad
 - Treat missing disk as an “Erasure”
- Today, disks so big that: RAID 5 not sufficient!
 - Time to repair disk sooooo long, another disk might fail in process!
 - “RAID 6” – allow 2 disks in replication stripe to fail
 - Requires more complex erasure code, such as EVENODD code (see readings)
- More general option for general erasure code: Reed-Solomon codes
 - Based on polynomials in $GF(2^k)$ (i.e. k-bit symbols)
 - m data points define a degree m polynomial; encoding is n points on the polynomial
 - Any m points can be used to recover the polynomial; $n - m$ failures tolerated
- Erasure codes not just for disk arrays. For example, geographic replication
 - E.g., split data into $m = 4$ chunks, generate $n = 16$ fragments and distribute across the Internet
 - Any 4 fragments can be used to recover the original data --- very durable!

Higher Durability through Geographic Replication

- Highly durable – hard to destroy all copies
- Highly available for reads
 - Simple replication: read any copy
 - Erasure coded: read m of n
- Low availability for writes
 - Can't write if any one replica is not up
 - Or – need relaxed consistency model
- Reliability? – availability, security, durability, fault-tolerance



Storage Reliability Problem

- Single logical file operation can involve updates to multiple physical disk blocks
 - inode, indirect block, data block, bitmap, ...
 - With sector remapping, single update to physical disk block can require multiple (even lower level) updates to sectors
- At a physical level, operations complete one at a time
 - Want concurrent operations for performance
- How do we guarantee consistency regardless of when crash occurs?

Threats to Reliability

- Interrupted Operation
 - Crash or power failure in the middle of a series of related updates may leave stored data in an inconsistent state
 - Example: transfer funds from one bank account to another
 - What if transfer is interrupted after withdrawal and before deposit?
- Loss of stored data
 - Failure of non-volatile storage media may cause previously stored data to disappear or be corrupted

Reliability Approach #1: Careful Ordering

- Sequence operations in a specific order
 - Careful design to allow sequence to be interrupted safely
 - Data block \leftarrow inode \leftarrow free \leftarrow directory
- Post-crash recovery
 - Read data structures to see if there were any operations in progress
 - Clean up/finish as needed
- Approach taken by
 - FAT and FFS (fsck) to protect filesystem structure/metadata
 - Many app-level recovery schemes (e.g., Word, emacs autosaves)

Berkeley FFS: Create a File

Normal operation:

- Allocate data block
- Write data block
- Allocate inode
- Write inode block
- Update bitmap of free blocks and inodes
- Update directory with file name → inode number
- Update modify time for directory

Recovery:

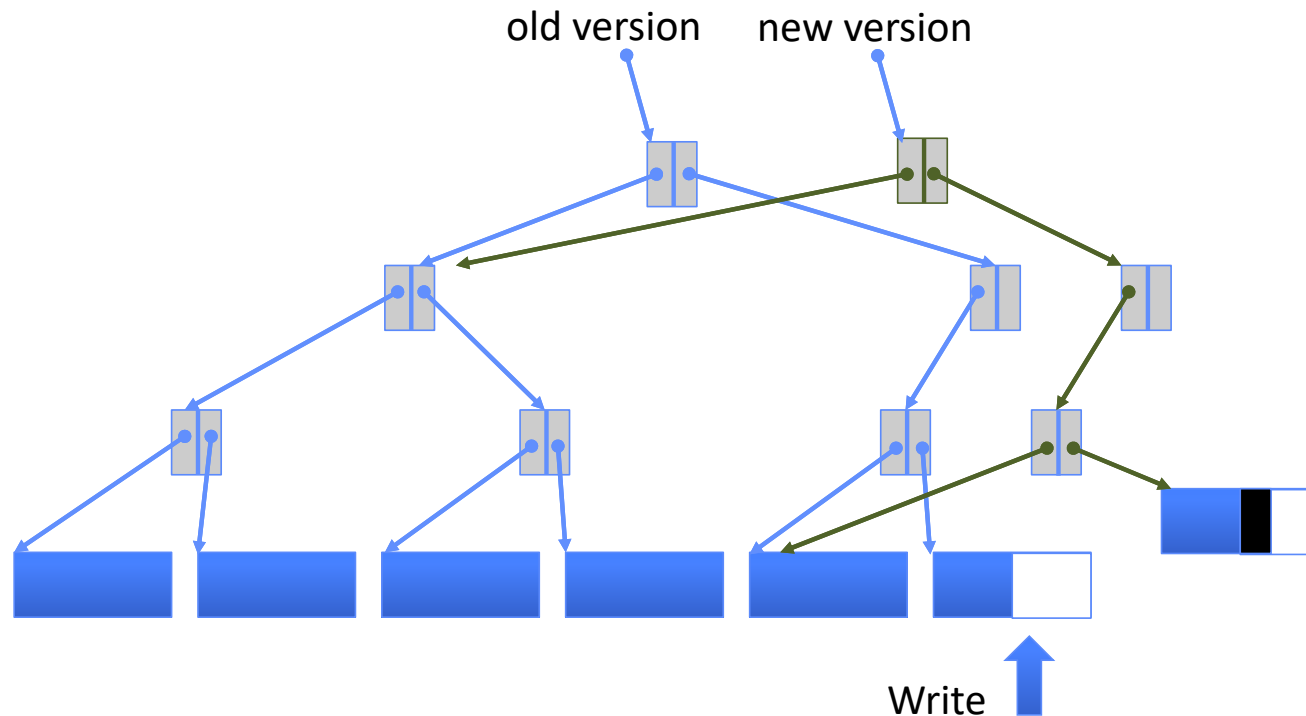
- Scan inode table
- If any unlinked files (not in any directory), delete or put in lost & found dir
- Compare free block bitmap against inode trees
- Scan directories for missing update/access times

Time proportional to disk size

Reliability Approach #2: Copy on Write File Layout

- Recall: multi-level index structure lets us find the data blocks of a file
- Instead of over-writing existing data blocks and updating the index structure:
 - Create a new version of the file with the updated data
 - Reuse blocks that don't change much of what is already in place
 - This is called: **Copy On Write (COW)**
- Seems expensive! But
 - Updates can be batched
 - Almost all disk writes can occur in parallel
- Approach taken in network file server appliances
 - NetApp's Write Anywhere File Layout (WAFL)
 - ZFS (Sun/Oracle) and OpenZFS

COW with Smaller-Radix Blocks



- If file represented as a tree of blocks, just need to update the leading fringe

Example: ZFS and OpenZFS

- Variable sized blocks: 512 B – 128 KB
- Symmetric tree
 - Know if it is large or small when we make the copy
- Store version number with pointers
 - Can create new version by adding blocks and new pointers
- Buffers a collection of writes before creating a new version with them
- Free space represented as tree of extents in each block group
 - Delay updates to freespace (in log) and do them all when block group is activated

More General Reliability Solutions

- Use Transactions for atomic updates
 - Ensure that multiple related updates are performed atomically
 - i.e., if a crash occurs in the middle, the state of the systems reflects either all or none of the updates
 - Most modern file systems use transactions internally to update filesystem structures and metadata
 - Many applications implement their own transactions
- Provide Redundancy for media failures
 - Redundant representation on media (Error Correcting Codes)
 - Replication across media (e.g., RAID disk array)

Transactions

- Closely related to critical sections for manipulating shared data structures
- They extend concept of atomic update from memory to stable storage
 - Atomically update multiple persistent data structures
- Many ad-hoc approaches
 - FFS carefully ordered the sequence of updates so that if a crash occurred while manipulating directory or inodes the disk scan on reboot would detect and recover the error (fsck)
 - Applications use temporary files and rename

Key Concept: Transaction

- A *transaction* is an atomic sequence of reads and writes that takes the system from consistent state to another.



- Recall: Code in a critical section appears atomic to other threads
- Transactions extend the concept of atomic updates from *memory* to *persistent storage*

Typical Structure

- **Begin** a transaction – get transaction id
- Do a bunch of updates
 - If any fail along the way, **roll-back**
 - Or, if any conflicts with other transactions, **roll-back**
- **Commit** the transaction

“Classic” Example: Transaction

```
BEGIN;      --BEGIN TRANSACTION
UPDATE accounts SET balance = balance - 100.00 WHERE
    name = 'Alice';

UPDATE branches SET balance = balance - 100.00 WHERE
    name = (SELECT branch_name FROM accounts WHERE name
    = 'Alice');

UPDATE accounts SET balance = balance + 100.00 WHERE
    name = 'Bob';

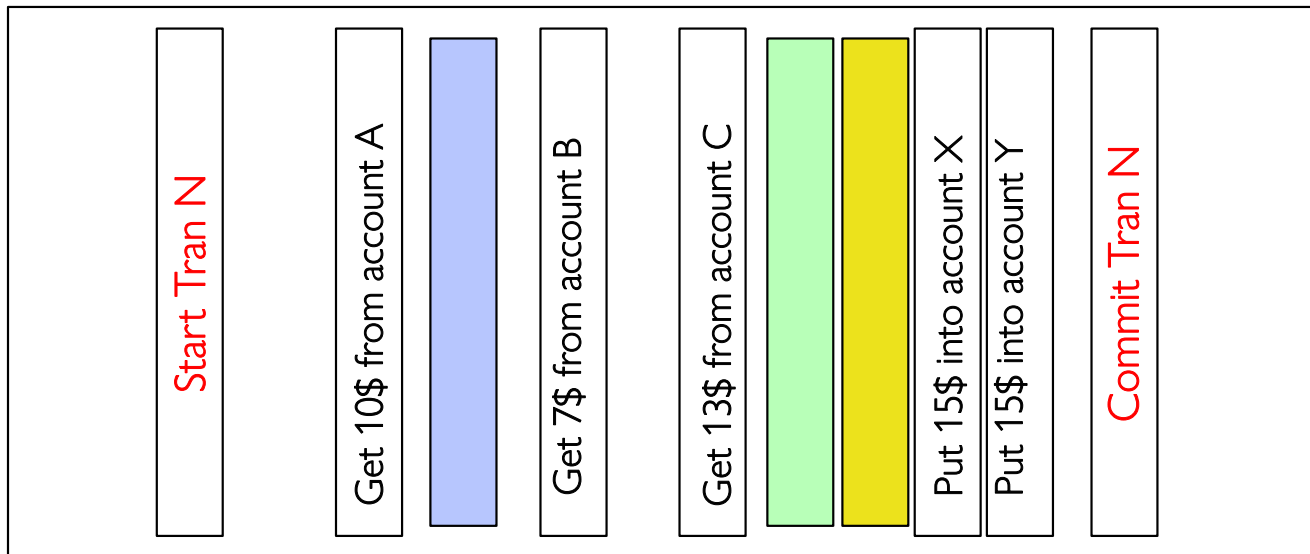
UPDATE branches SET balance = balance + 100.00 WHERE
    name = (SELECT branch_name FROM accounts WHERE name
    = 'Bob');

COMMIT;     --COMMIT WORK
```

Transfer \$100 from Alice's account to Bob's account

Concept of a log

- One simple action is atomic – write/append a basic item
- Use that to seal the commitment to a whole series of actions



Transactional File Systems

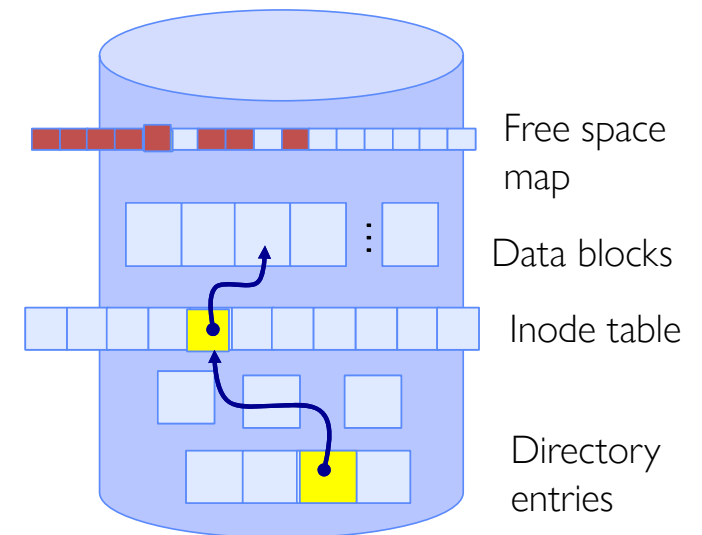
- Better reliability through use of log
 - Changes are treated as transactions
 - A transaction is committed once it is written to the log
 - » Data forced to disk for reliability
 - » Process can be accelerated with NVRAM
 - Although File system may not be updated immediately, data preserved in the log
- Difference between “Log Structured” and “Journaled”
 - In a Log Structured filesystem, data stays in log form
 - In a Journaled filesystem, Log used for recovery

Journaling File Systems

- Don't modify data structures on disk directly
- Write each update as transaction recorded in a log
 - Commonly called a journal or intention list
 - Also maintained on disk (allocate blocks for it when formatting)
- Once changes are in the log, they can be safely applied to file system
 - e.g. modify inode pointers and directory mapping
- Garbage collection: once a change is applied, remove its entry from the log
- Linux took original FFS-like file system (ext2) and added a journal to get ext3!
 - Some options: whether or not to write all data to journal or just metadata
- Other examples: NTFS, Apple HFS+/apfs, Linux XFS, JFS, ext4

Creating a File (No Journaling Yet)

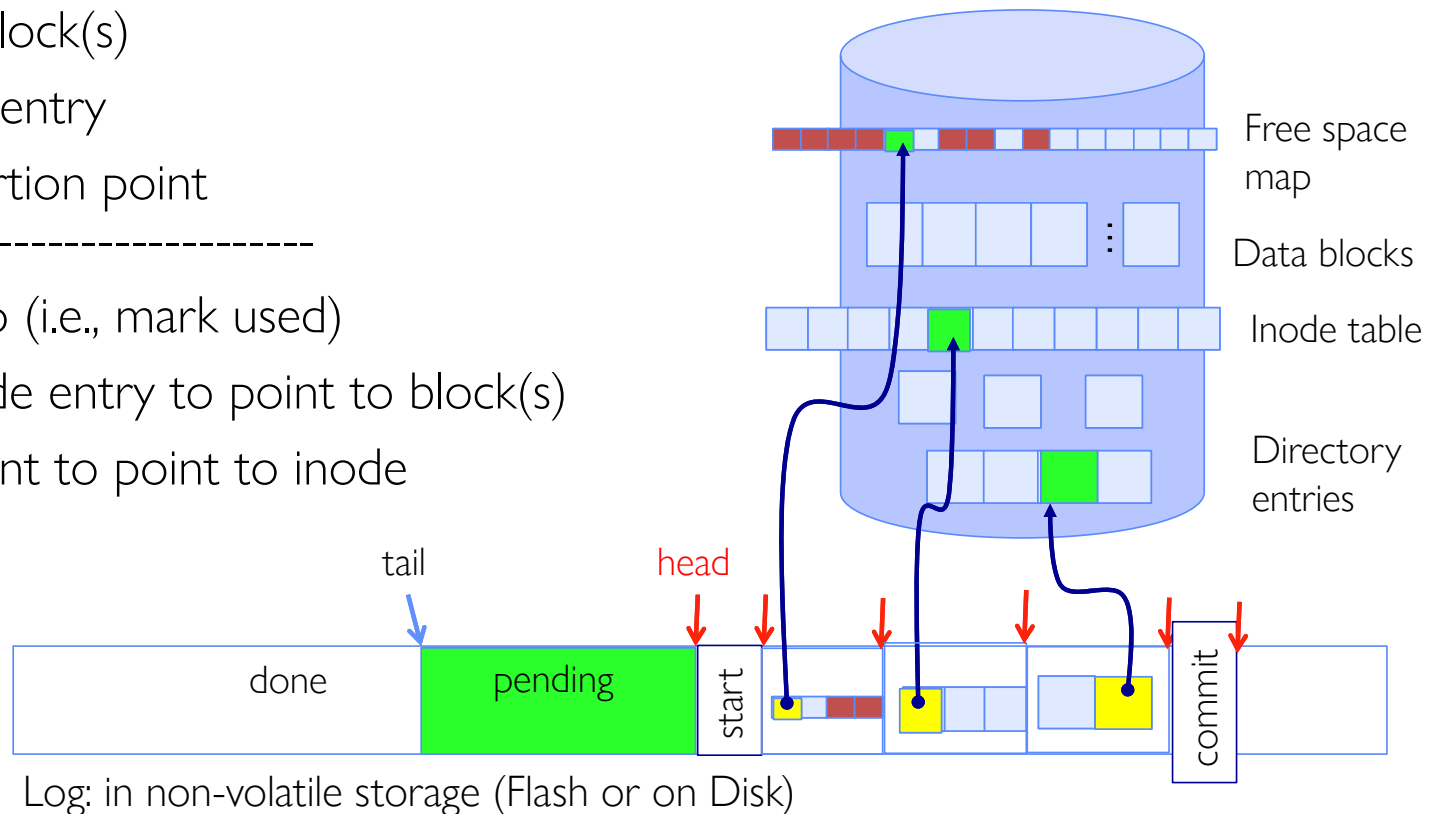
- Find free data block(s)
 - Find free inode entry
 - Find dirent insertion point
-
- Write map (i.e., mark used)
 - Write inode entry to point to block(s)
 - Write dirent to point to inode



Creating a File (With Journaling)

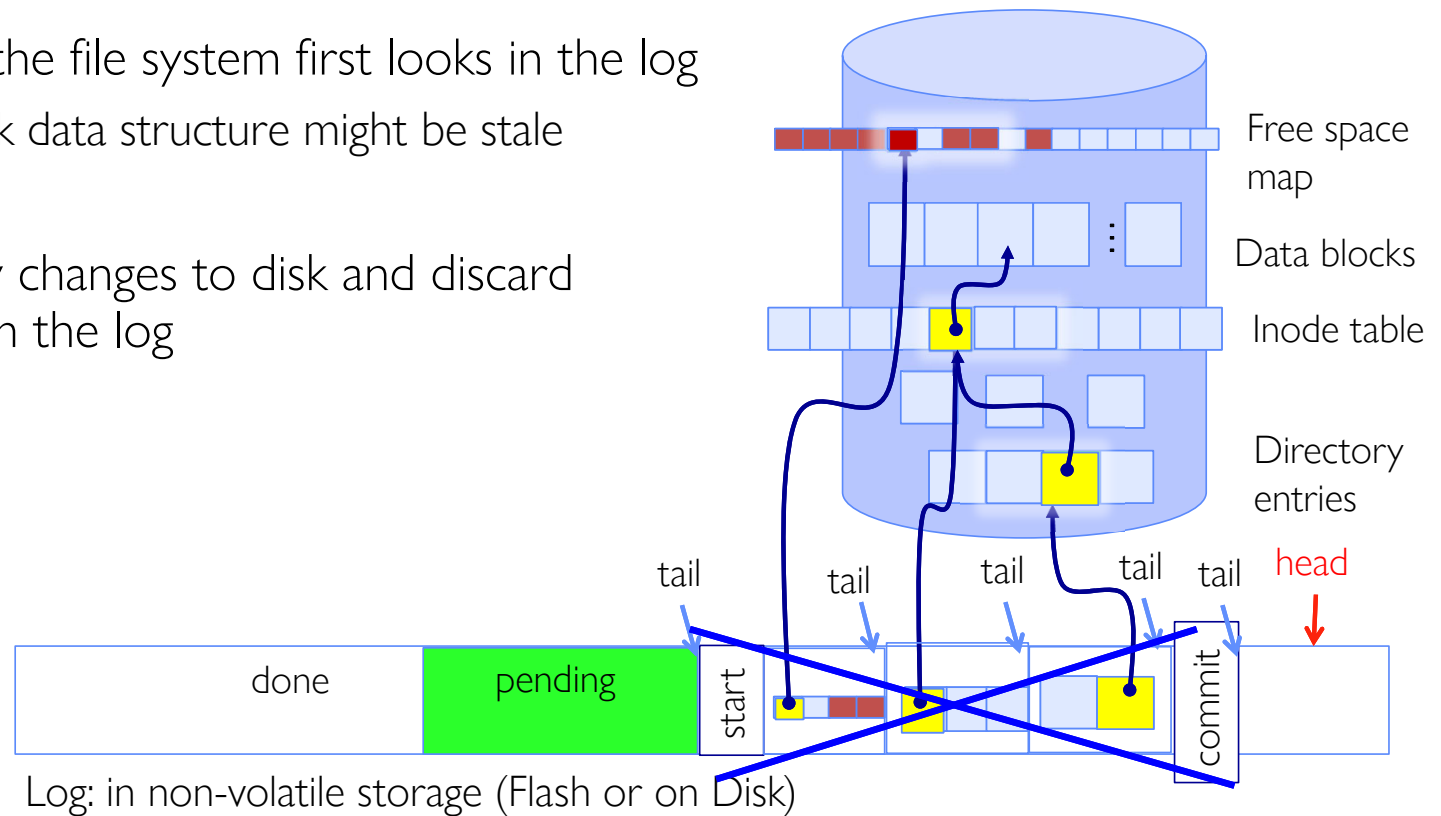
- Find free data block(s)
- Find free inode entry
- Find dirent insertion point

-
- [log] Write map (i.e., mark used)
 - [log] Write inode entry to point to block(s)
 - [log] Write dirent to point to inode



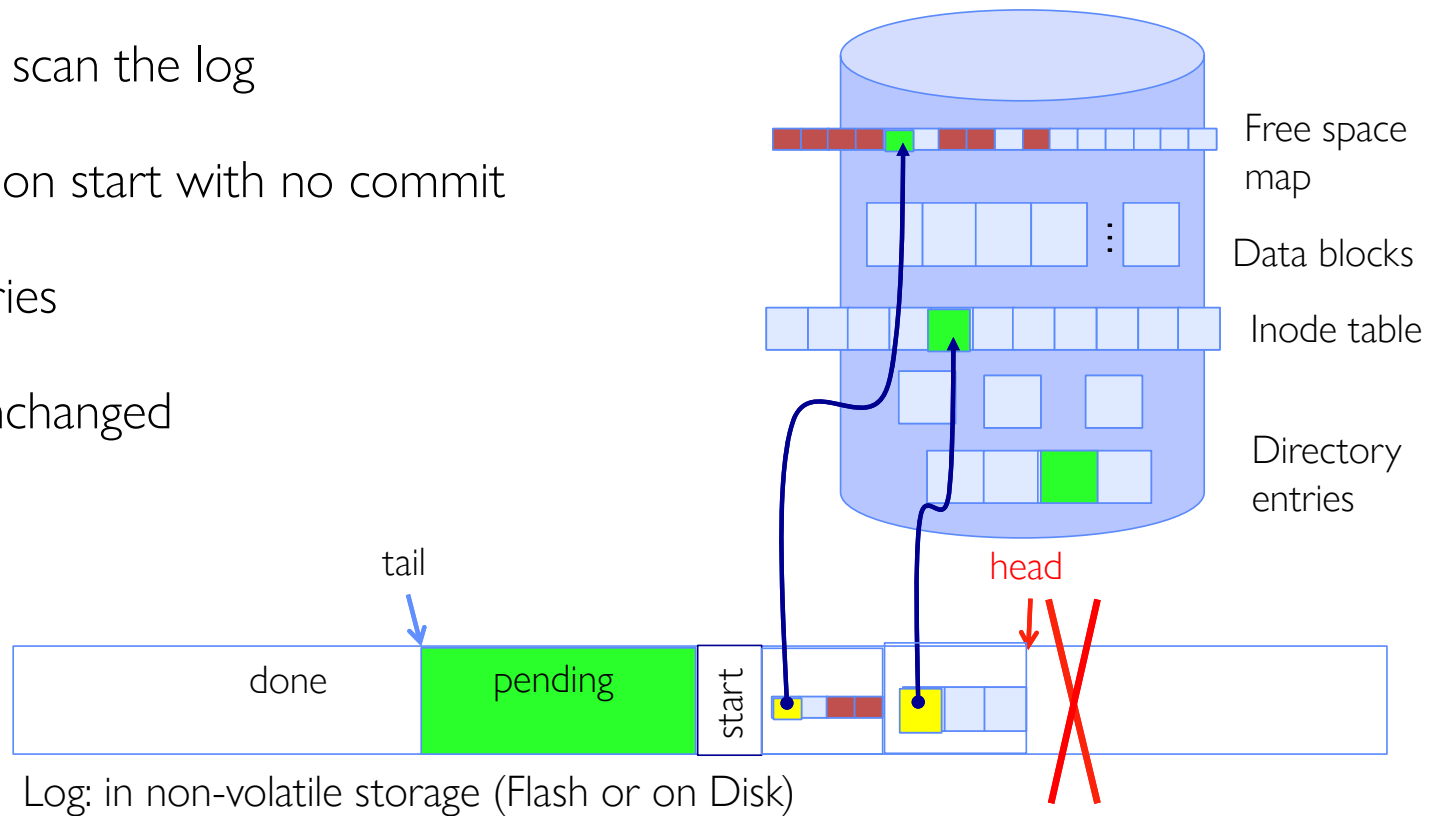
After Commit, Eventually Replay Transaction

- All accesses to the file system first looks in the log
 - Actual on-disk data structure might be stale
- Eventually, copy changes to disk and discard transaction from the log



Crash Recovery: Discard Partial Transactions

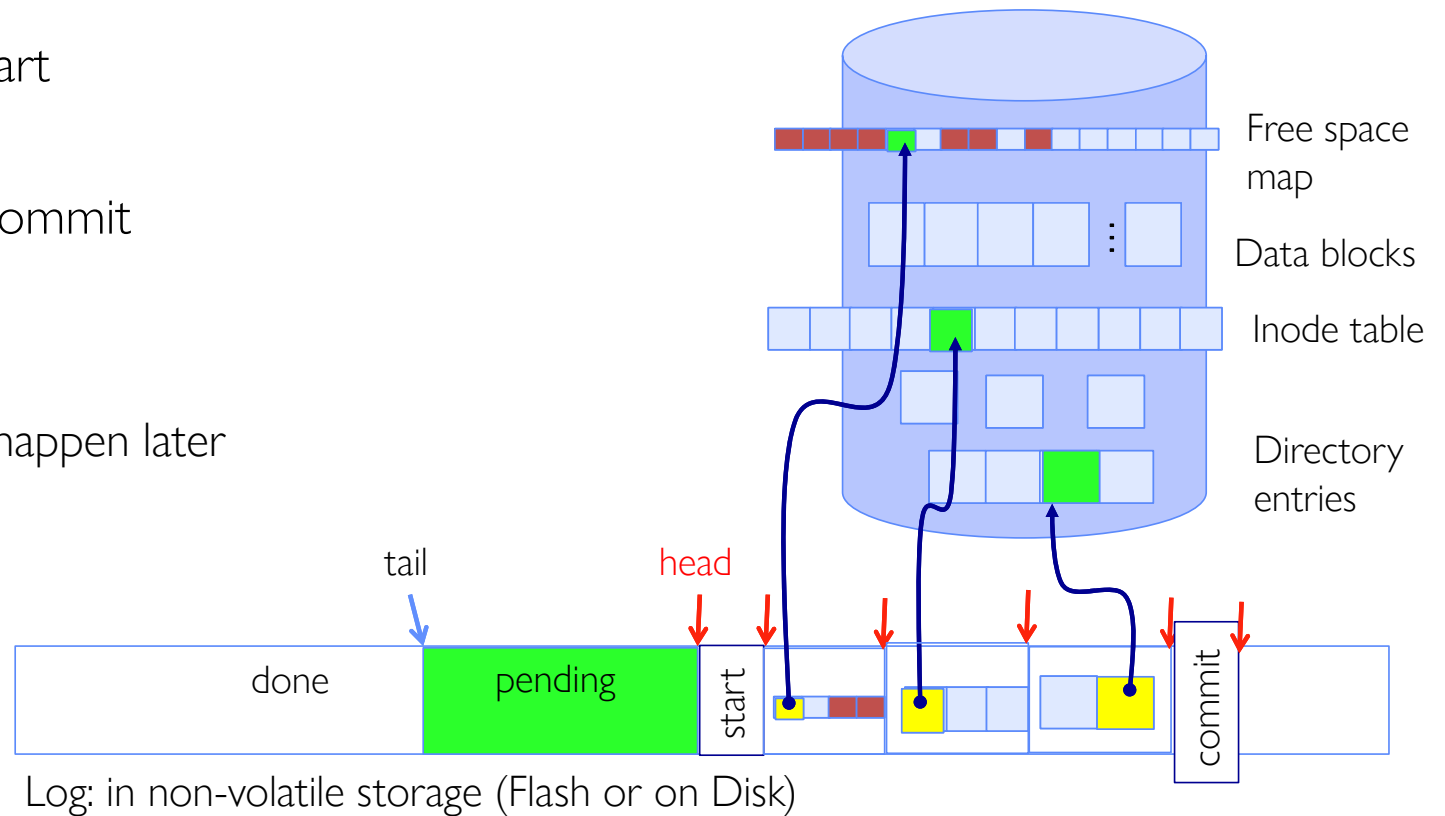
- Upon recovery, scan the log
- Detect transaction start with no commit
- Discard log entries
- Disk remains unchanged



Log: in non-volatile storage (Flash or on Disk)

Crash Recovery: Keep Complete Transactions

- Scan log, find start
- Find matching commit
- Redo it as usual
 - Or just let it happen later



Journaling Summary

Why go through all this trouble?

- Updates atomic, even if we crash:
 - Update either gets fully applied or discarded
 - All physical operations *treated as a logical unit*

Isn't this expensive?

- Yes! We're now writing all data twice (once to log, once to actual data blocks in target file)
- Modern filesystems journal metadata updates only
 - Record modifications to file system data structures
 - But apply updates to a file's contents directly

Conclusion

- Buffer Cache: Memory used to cache kernel resources, including disk blocks and name translations
 - Can contain “dirty” blocks (blocks yet on disk)
- Deep interactions between mem management, file system, sharing
 - **mmap()**: map file or anonymous segment to memory
- RAID and Erasure coding
 - Provide durability through coding and replication
- Copy-on-write provides richer function (versions) with much simpler recovery
 - Little performance impact since sequential write to storage device is nearly free
- Transactions over a log provide a general solution
 - Journalled file systems such as ext3, NTFS
 - Commit sequence to durable log, then update the disk
 - Log takes precedence over disk
 - Replay committed transactions, discard partials