Operating Systems
(Honor Track)

File System 3: Buffering, Reliability, and Transactions

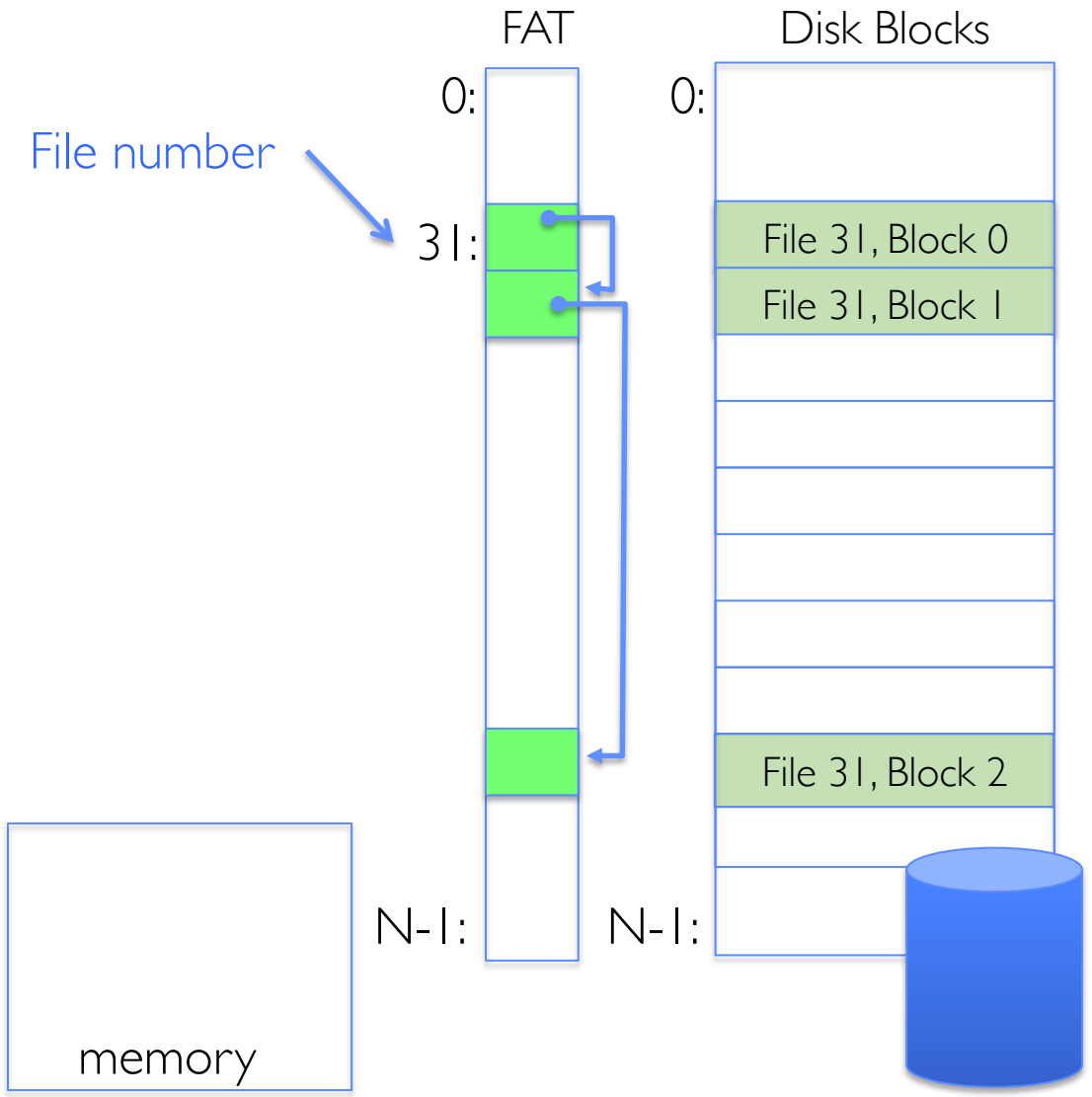
Xin Jin

Spring 2026

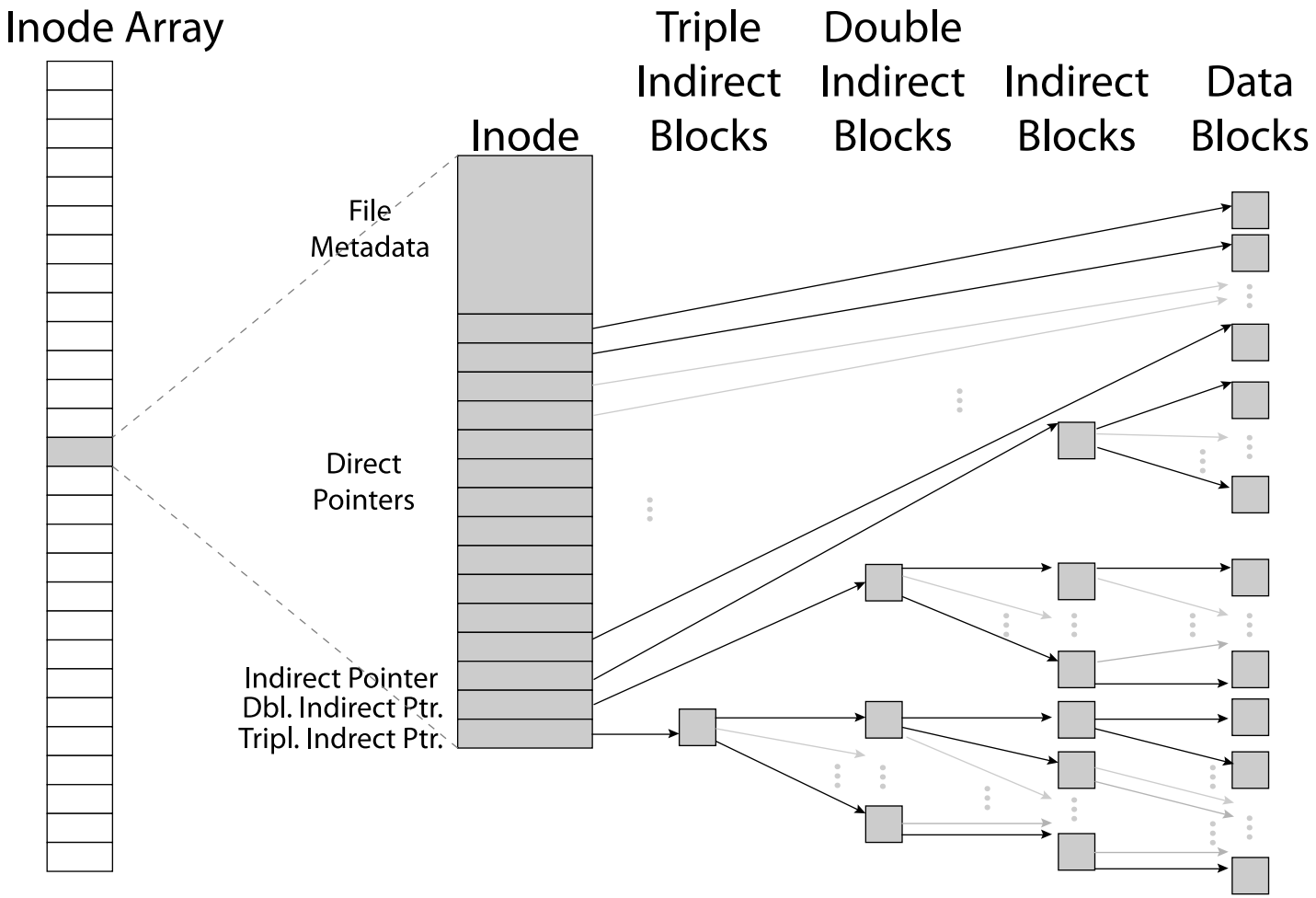
Acknowledgments: Ion Stoica, Berkeley CS 162

Recap: FAT (File Allocation Table)

- Assume (for now) we have a way to translate a path to a "file number"
  - i.e., a directory structure
- Disk Storage is a collection of Blocks
  - Just hold file data (offset o = < B, x >)
- Example: file_read 31, < 2, x >
  - Index into FAT with file number
  - Follow linked list to block
  - Read the block from disk into memory

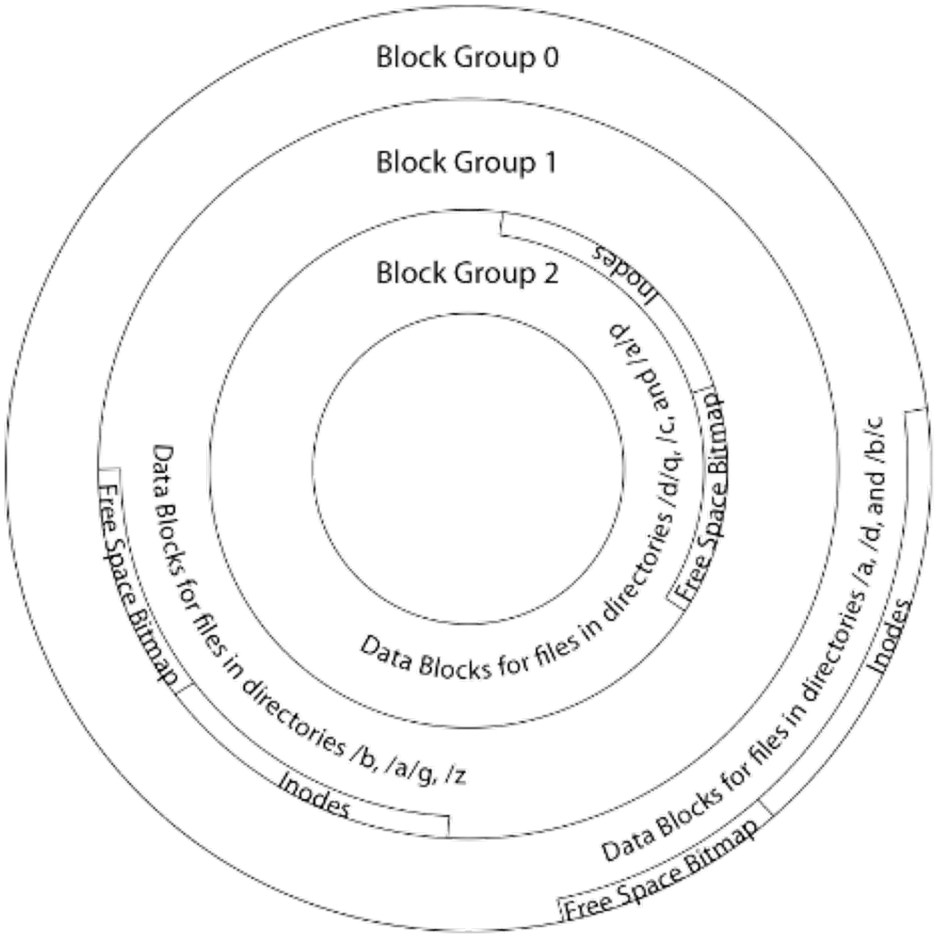


Recap: Inode Structure

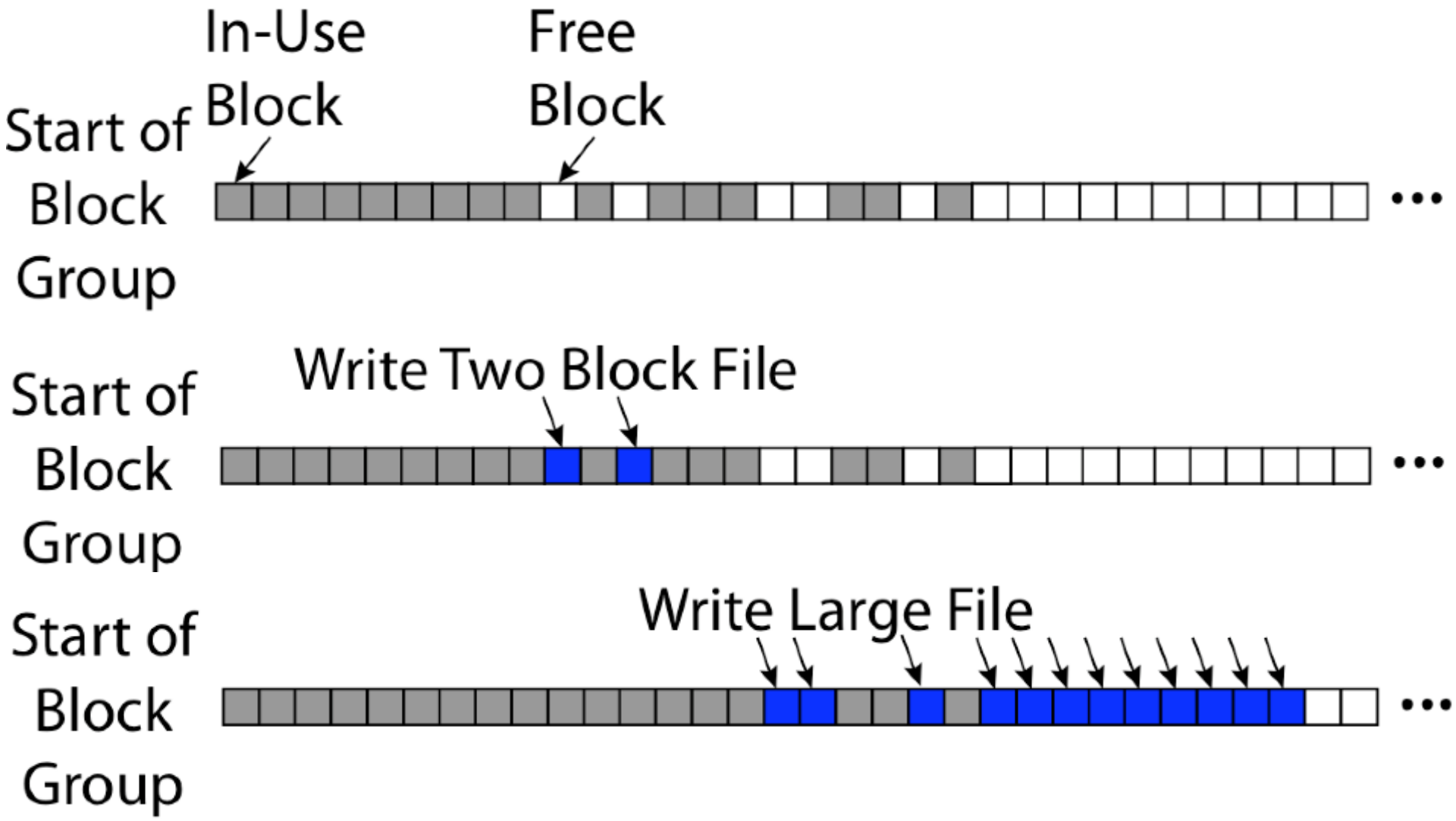


Recap: FFS Locality: Block Groups

- The UNIX BSD 4.2 (FFS) distributed the header information (inodes) closer to the data blocks
  - Often, inode for file stored in same “cylinder group” as parent directory of the file
  - makes an “ls” of that directory run very fast
- File system volume divided into set of block groups
  - Close set of tracks
- Data blocks, metadata, and free space interleaved within block group
  - Avoid huge seeks between user data and system structure
- Put directory and its files in common block group

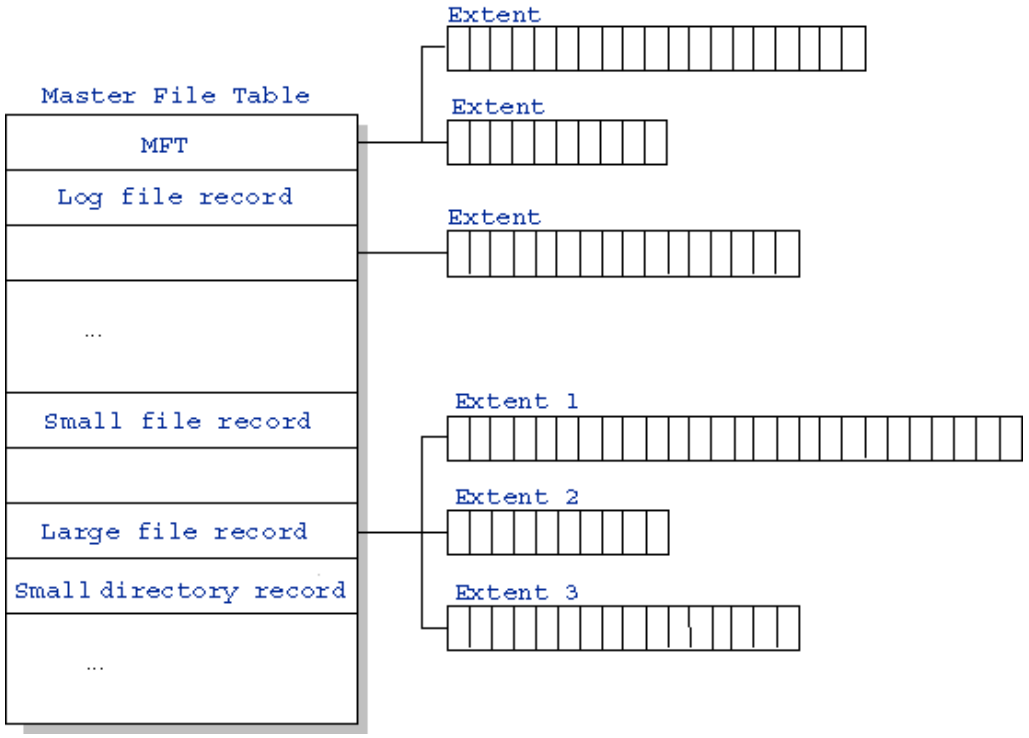


Recap: UNIX 4.2 BSD FFS First Fit Block Allocation



Recap: New Technology File System (NTFS)

- Default on modern Windows systems
- Variable length extents
  - Rather than fixed blocks
- Instead of FAT or inode array: Master File Table
  - Like a database, with max 1 KB size for each table entry
  - Everything (almost) is a sequence of <attribute:value> pairs
    - » Meta-data and data
- Each entry in MFT contains metadata and:
  - File's data directly (for small files)
  - A list of *extents* (start block, size) for file's data
  - For big files: pointers to other MFT entries with *more* extent lists

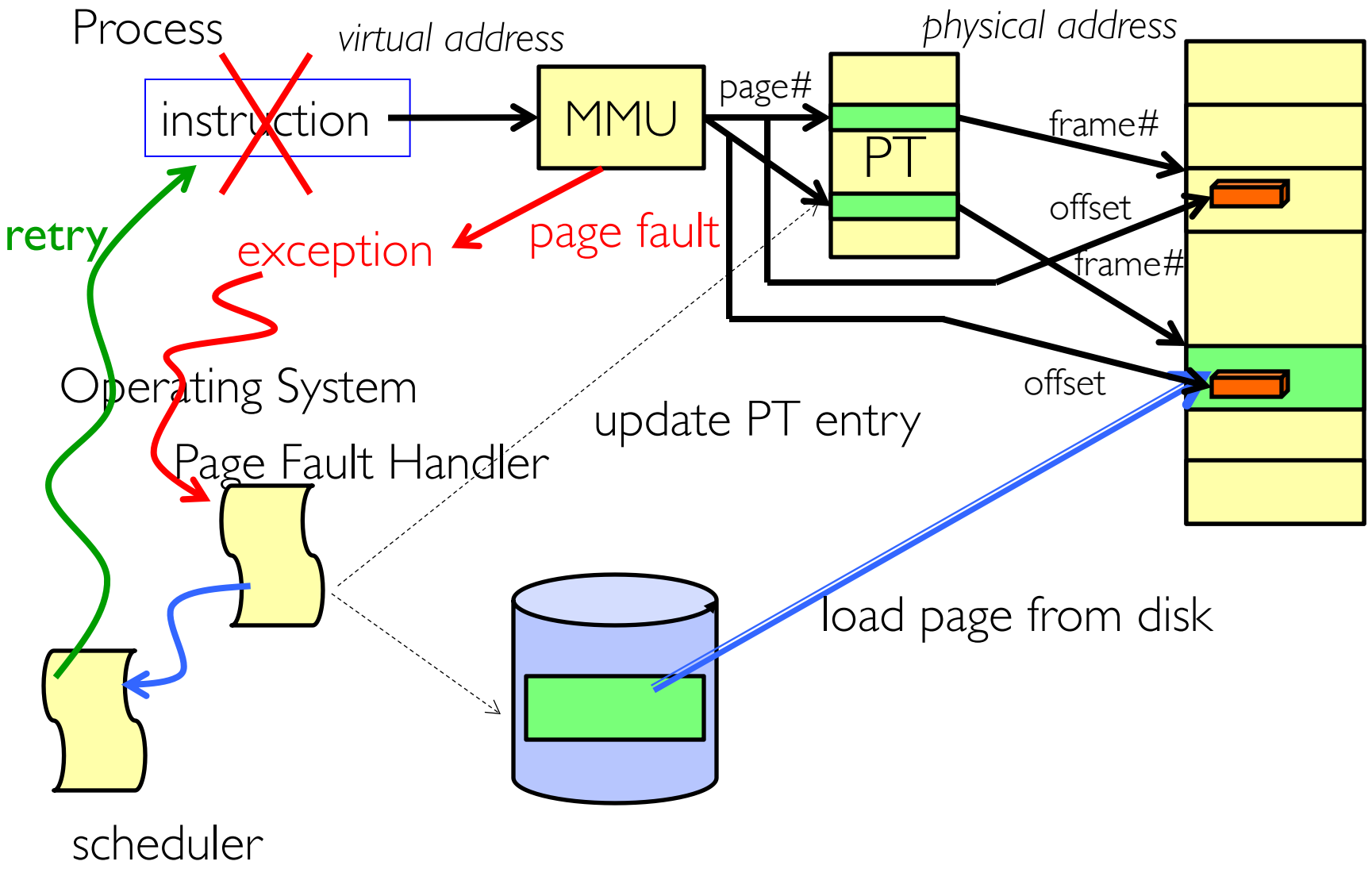


MEMORY MAPPED FILES

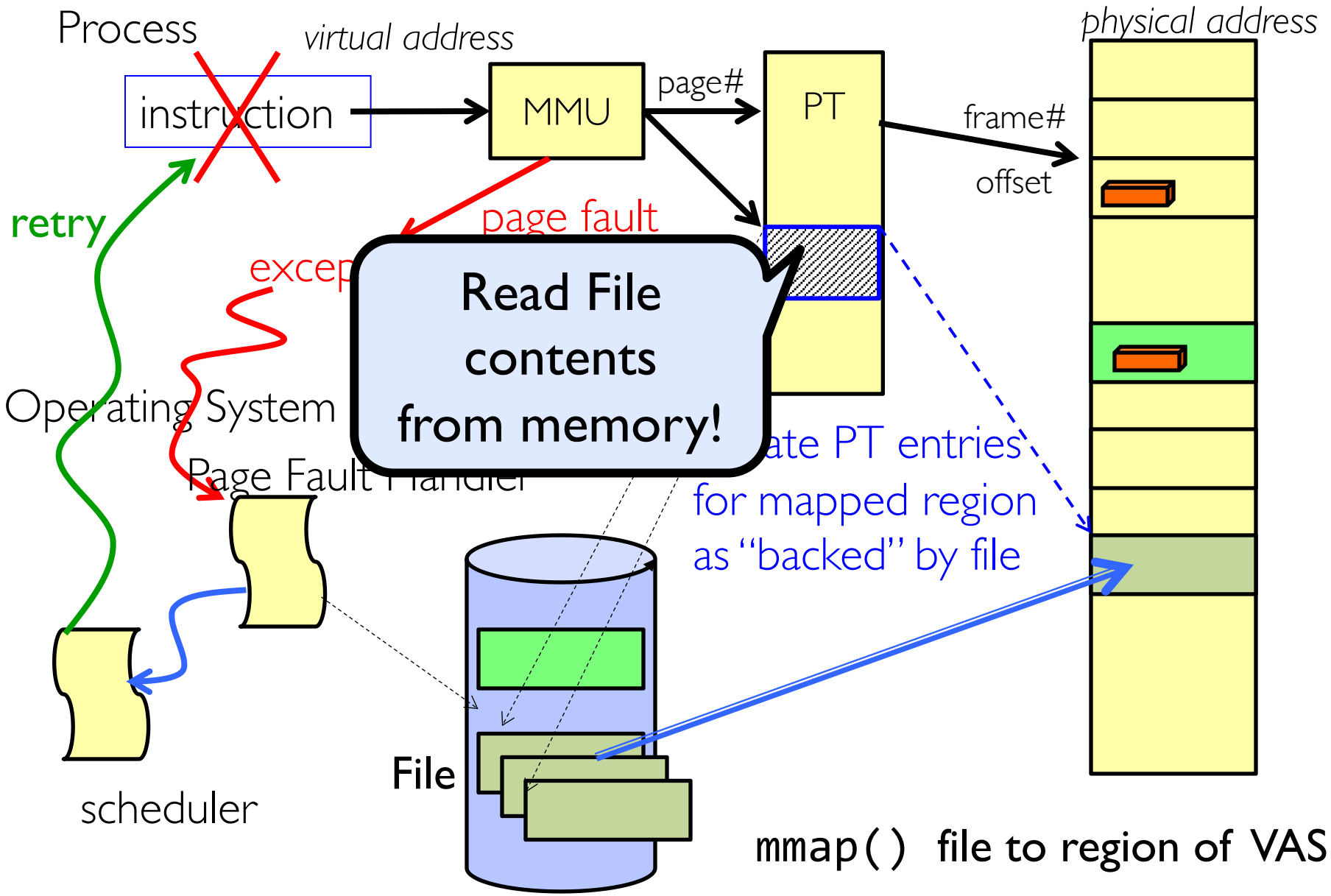
Memory Mapped Files

- Traditional I/O involves explicit transfers between buffers in process address space to/from regions of a file
  - This involves multiple copies into buffers 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
- 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
     mmap -- allocate memory, or map files or devices into memory

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <sys/mman.h>

     void *
     mmap(void *addr, size_t len, int prot, int flags, int fd,
         off_t offset);

DESCRIPTION
     The mmap() system call causes the pages starting at addr and continuing
     for at most len bytes to be mapped from the object described by fd,
     starting at byte offset offset.  If offset or len is not a multiple of
```

- 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 unsigned int) &something);
  printf("Heap at : %16lx\n", (long unsigned int) malloc(1));
  printf("Stack at: %16lx\n", (long unsigned int) &mfile);

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

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

  printf("mmap returned: %16lx\n", (long unsigned int) mfile);

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

Return starting address

OS chooses starting address

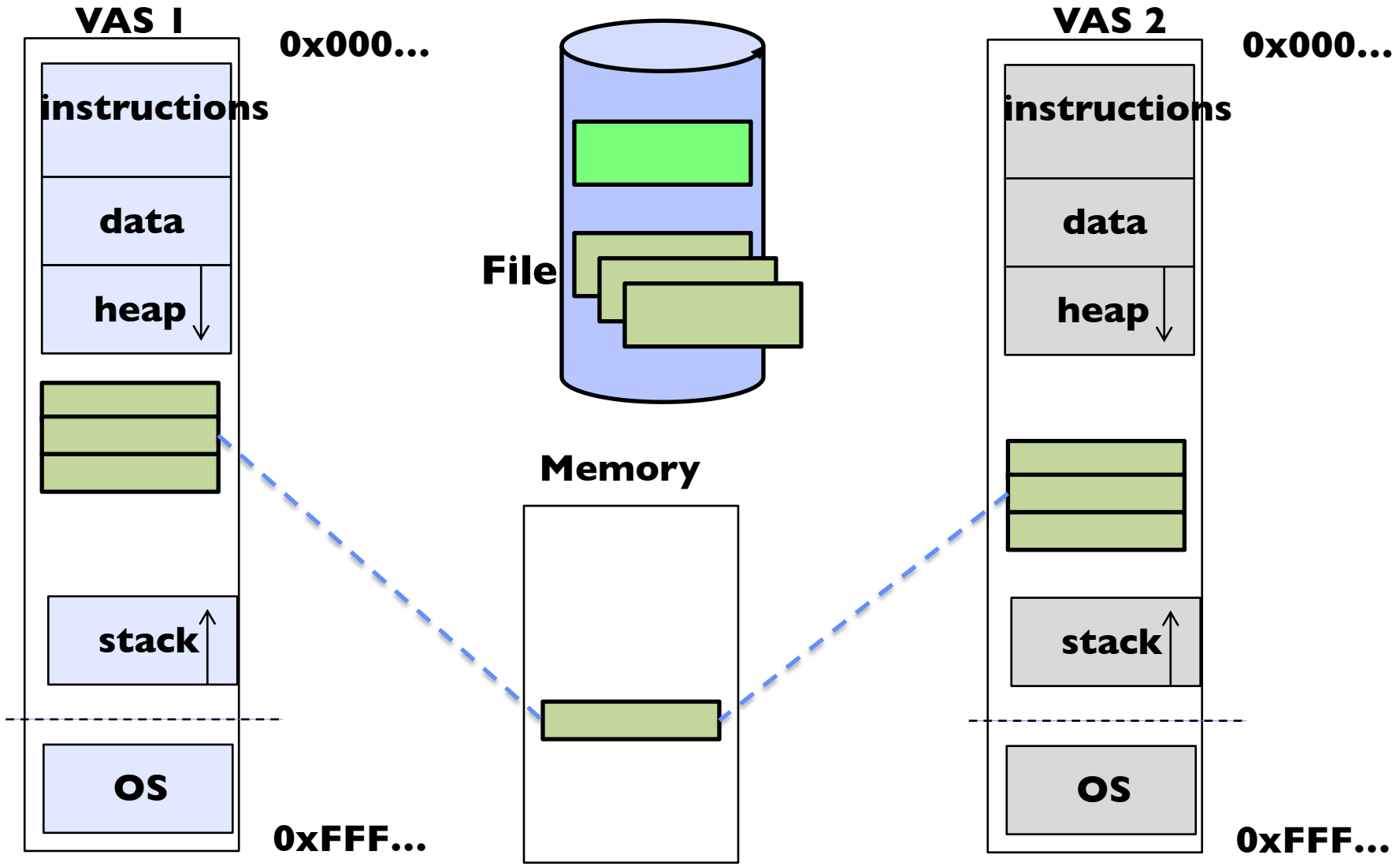
An mmap( ) Example

```
#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
  printf("Heap at : %16lx\n", (long
  printf("Stack at: %16lx\n", (long

  /* Open the file */
  myfd = open(argv[1], O_RDWR | O_CR
  if (myfd < 0) { perror("open failed!

  /* map the file */
  mfile = mmap(0, 10000, PROT_READ|P
  if (mfile == MAP_FAILED) {perror(

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

  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
ThiLet'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

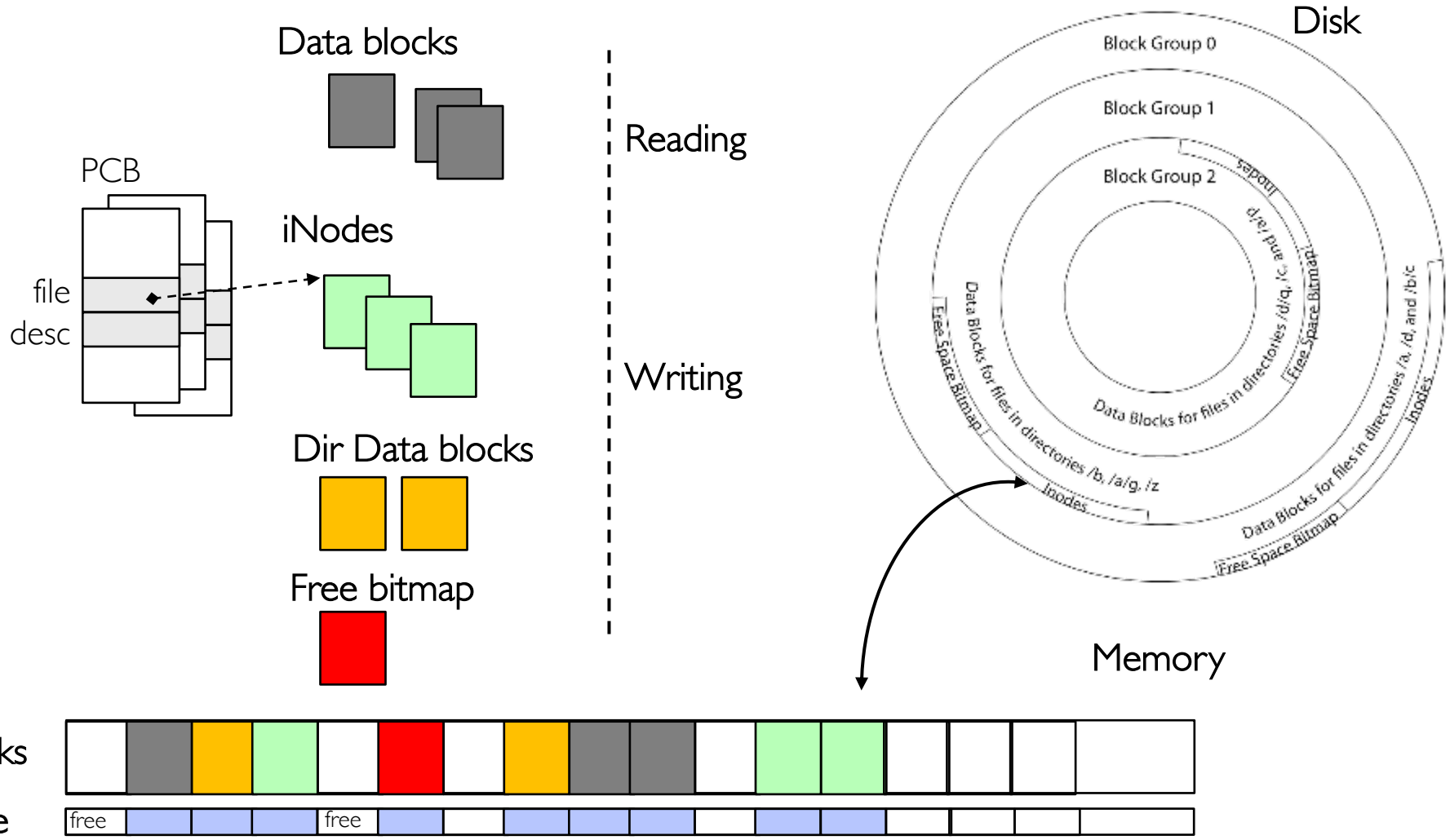
THE BUFFER CACHE

Buffer Cache

- Kernel *must* copy disk blocks to main memory to access their contents and write them back if modified
  - Could be data blocks, inodes, directory contents, etc.
  - Possibly dirty (modified and not 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)

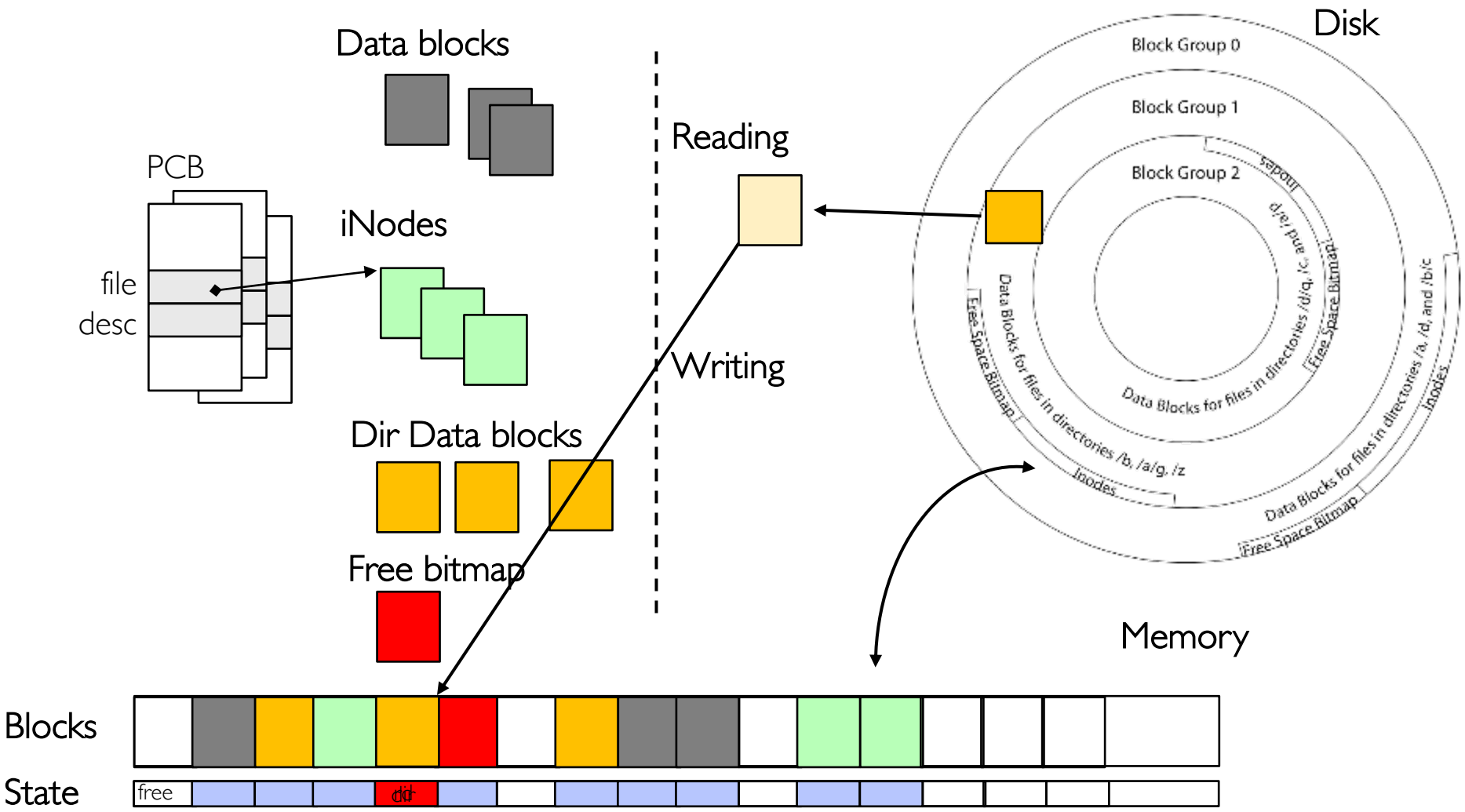
File System Buffer Cache

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



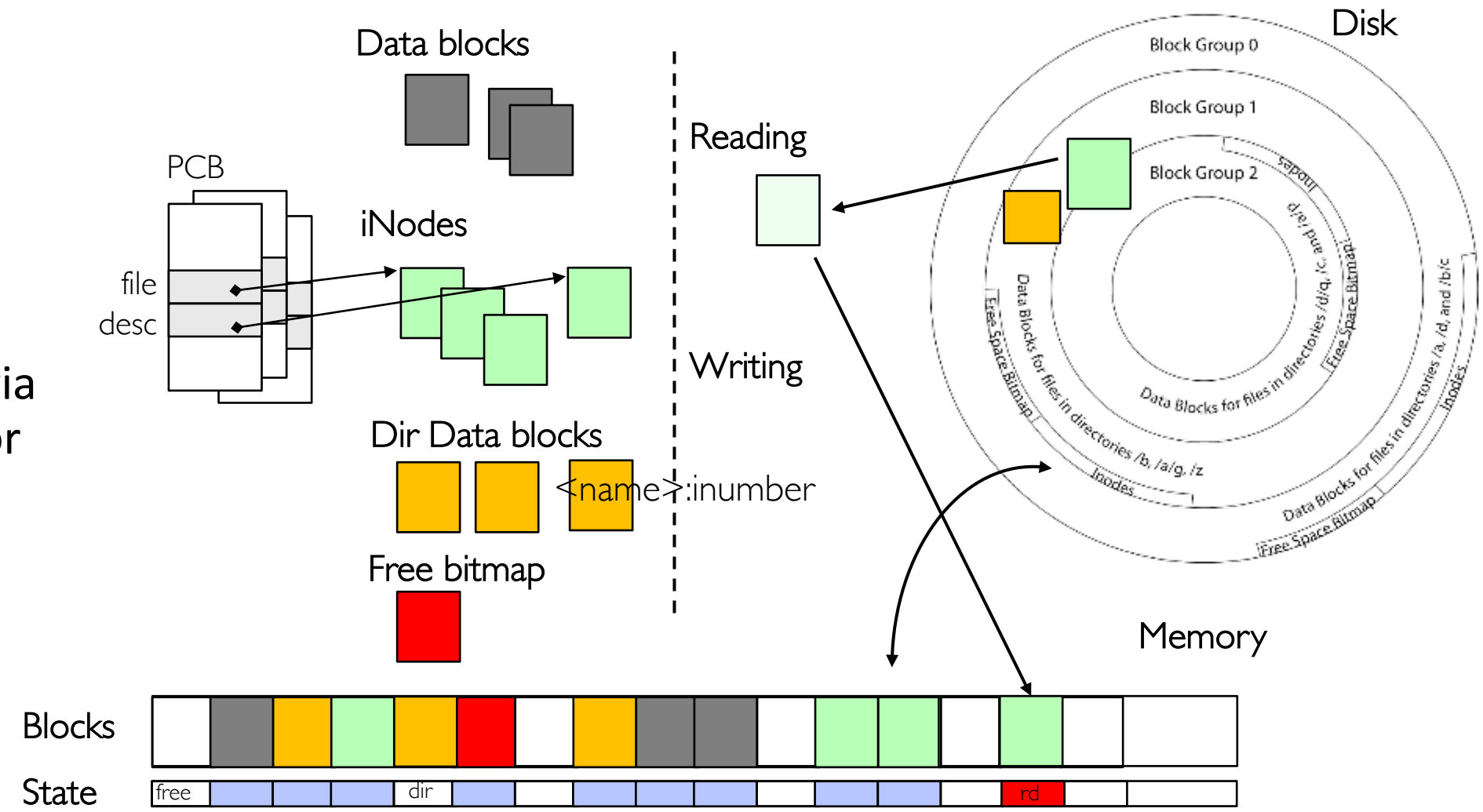
File System Buffer Cache: open

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



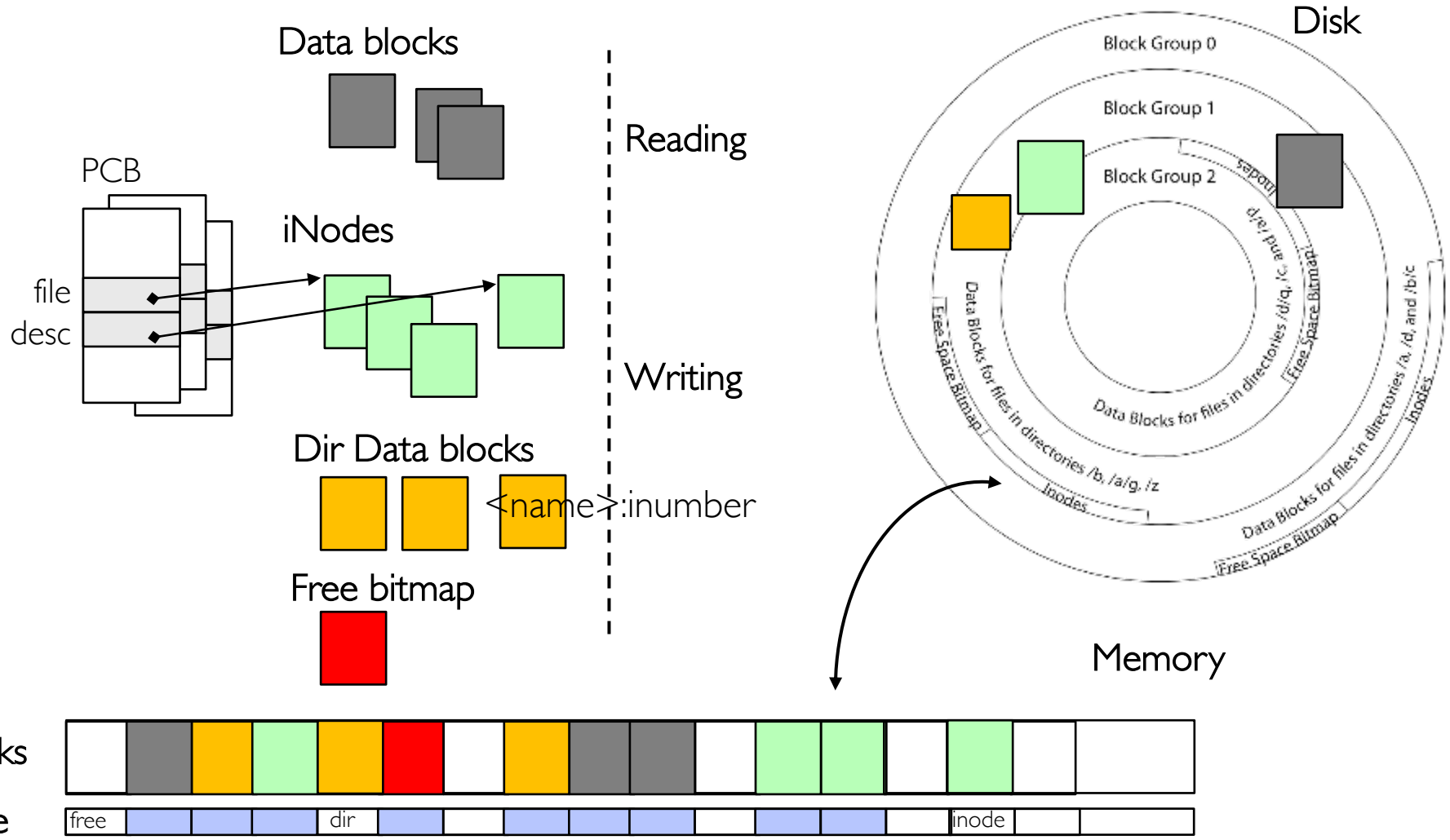
File System Buffer Cache: open

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



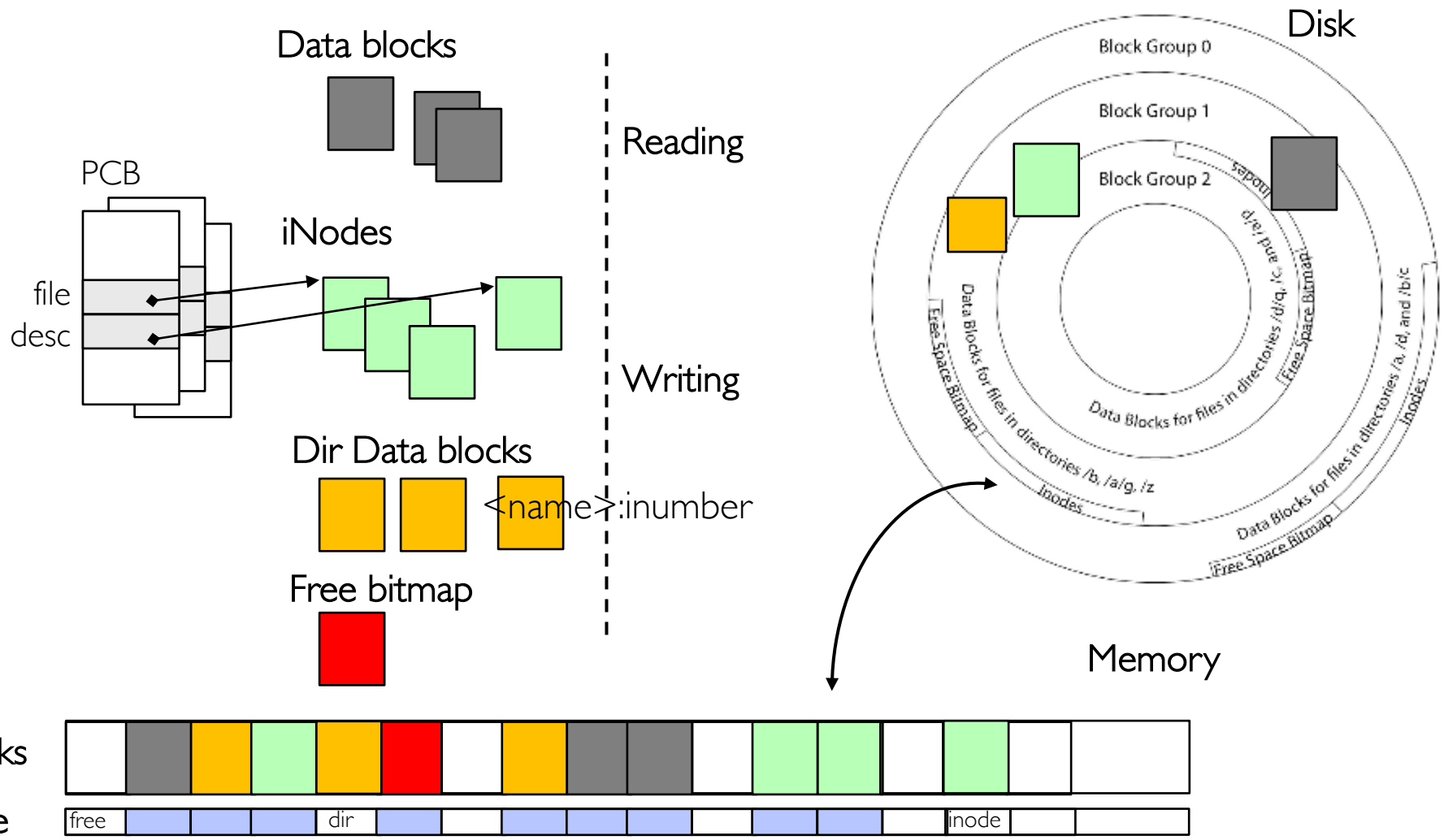
File System Buffer Cache: Read?

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



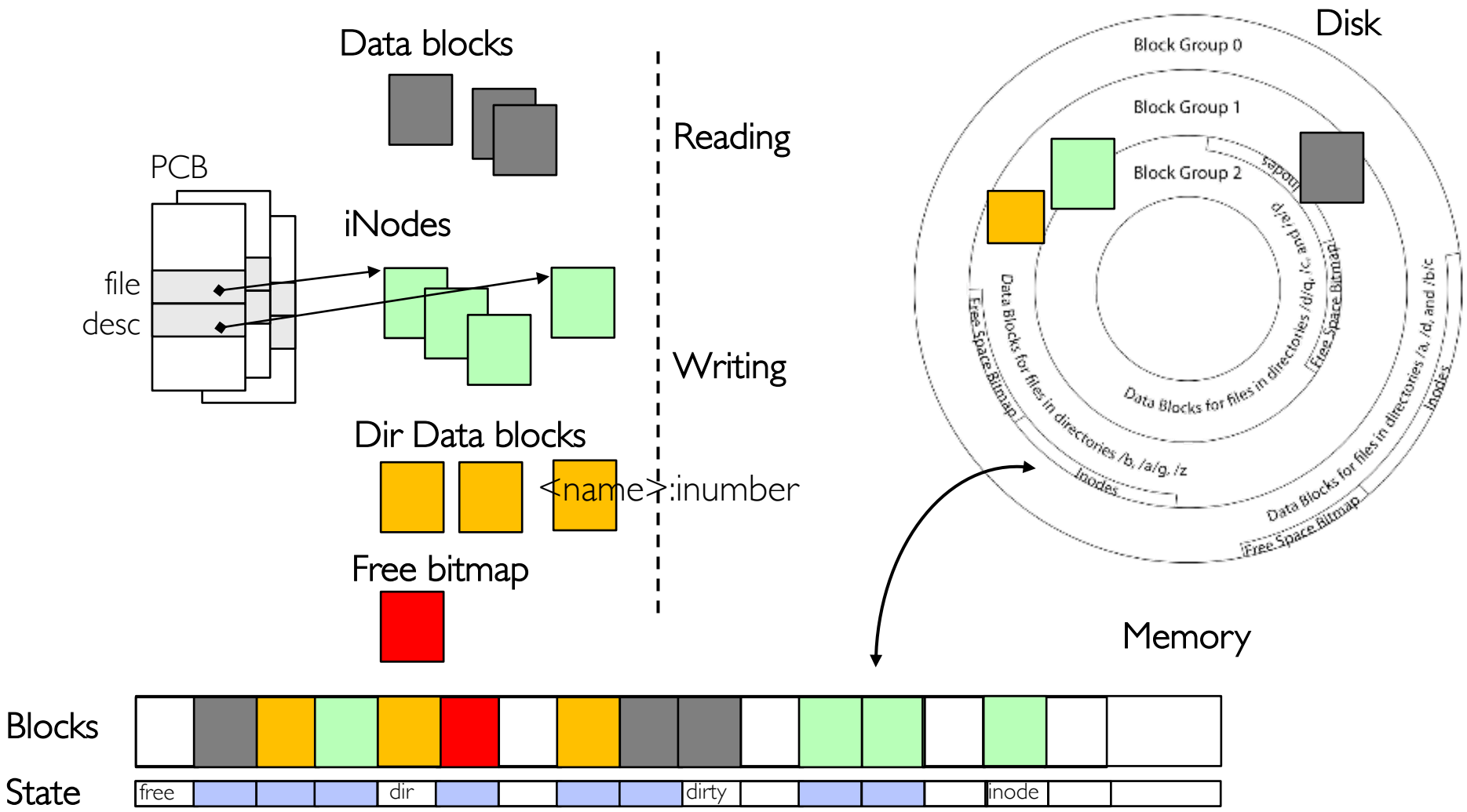
File System Buffer Cache: Write?

- 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
- Blocks are used for a variety of purposes
  - inodes, data for dirs and files, freemap
  - OS maintains pointers into them
- Termination – e.g., process exit – open, read, write
- Replacement – what to do when it fills up?

File System Caching

- Replacement policy?
  - LRU. Can afford overhead for full LRU implementation.
  - Advantages:
    - » 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

- 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 ⇒ won’t be able to run many applications
  - Too little memory to file system cache ⇒ 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!

Group Discussion

- Topic: Buffer Caching and Demand Paging
  - Can you compare buffer caching and demand paging?

- Discuss in groups of two to three students
  - Each group chooses a leader to summarize the discussion
  - In your group discussion, please do not dominate the discussion, and give everyone a chance to speak

Buffer Caching vs. Demand Paging

- Demand paging
  - LRU is infeasible; use approximation (like Clock)
  - Evict not-recently-used pages when memory is close to full

- Buffer Cache
  - LRU is OK
  - 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

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: data in disk is durable, but cannot be accessed when the machine is down

- 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*?

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

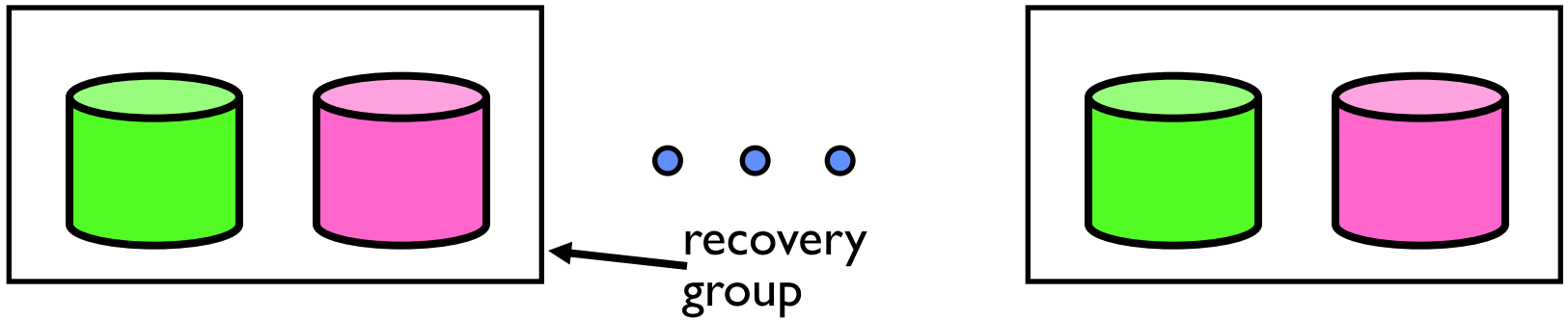
- Make sure writes survive in short term
  - Either abandon delayed writes or
  - Use special, battery-backed RAM (called non-volatile RAM or NVRAM) for dirty blocks in buffer cache

- 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…

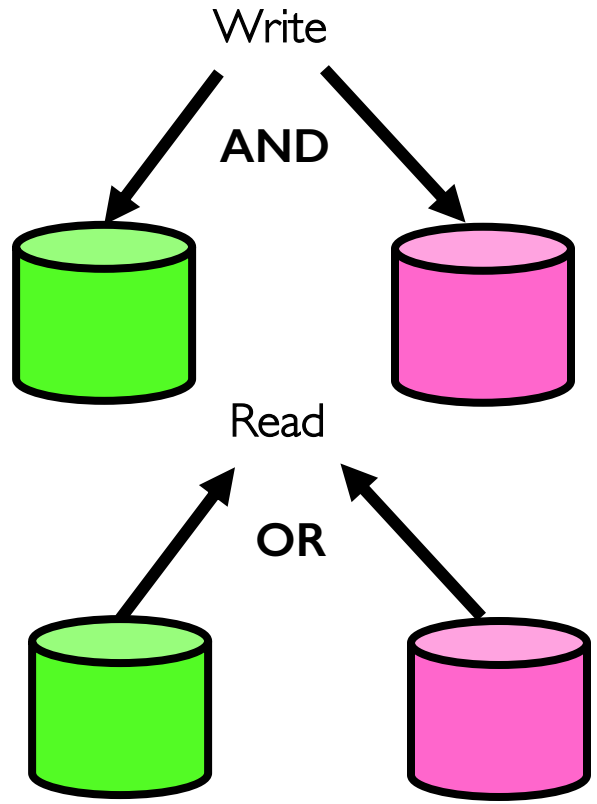
RAID

- RAID: Redundant array of inexpensive/independent disks
- Goal: reliability, performance, capacity
- Data storage virtualization
  - Build a logical disk drive from multiple physical disk drives
  - Better reliability, performance and capacity than a single physical drive

RAID 1: Disk Mirroring/Shadowing

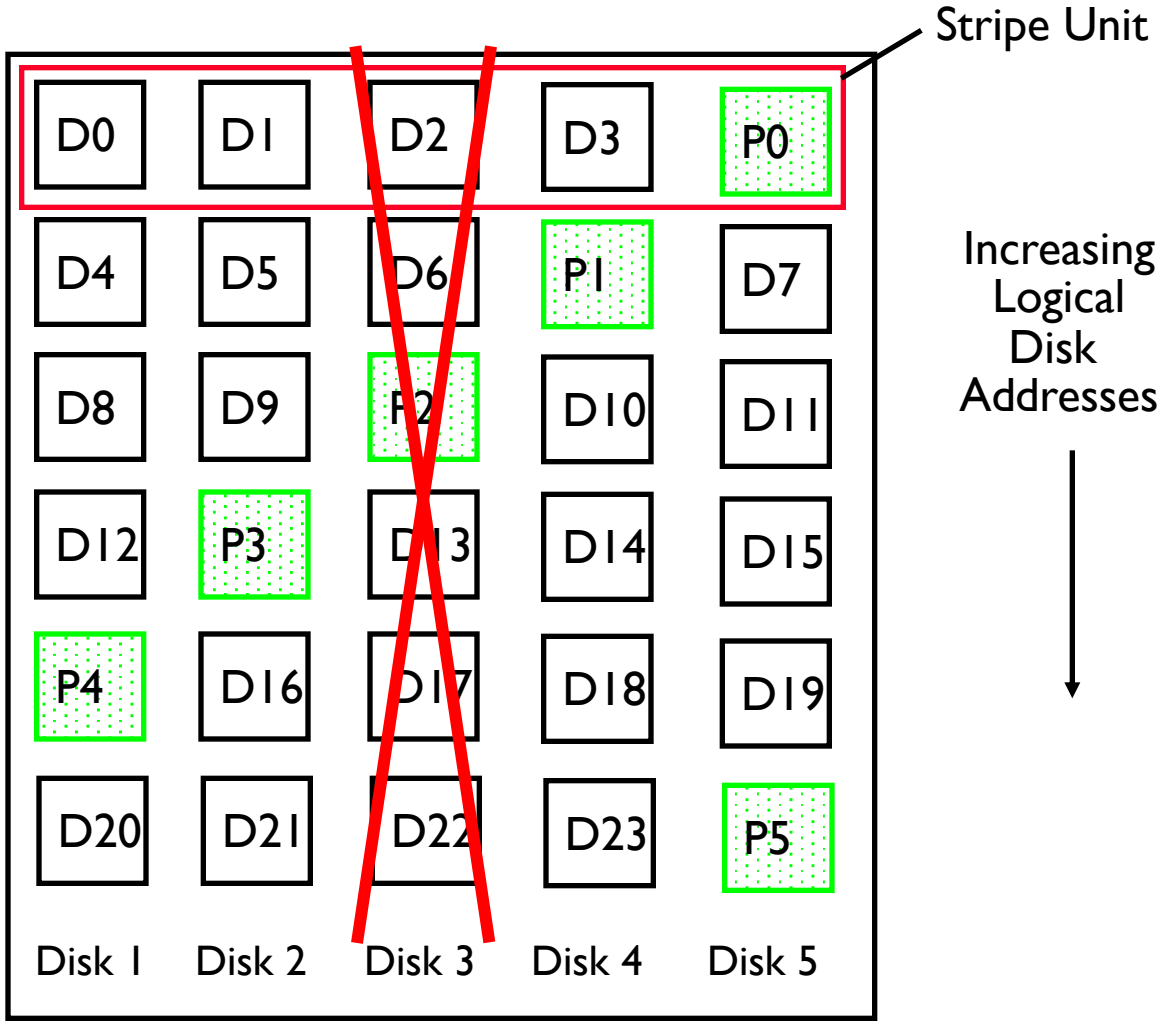


- Each disk is fully duplicated onto its "shadow"
  - For high I/O rate, high availability environments
  - Most expensive solution: 100% capacity overhead
- Bandwidth sacrificed on write:
  - Logical write = two physical writes
  - Highest bandwidth when disk heads and rotation synchronized (challenging)
- Reads may be optimized
  - Can have two independent reads to same data
- Recovery:
  - Disk failure ⇒ 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 stripped across multiple disks
  - Successive blocks stored on successive (non-parity) disks
  - Increased bandwidth over single disk
- Parity block (in green) constructed by XORing data bocks in stripe
  - P0=D0⊕D1⊕D2⊕D3
  - Can destroy any one disk and still reconstruct data
- Suppose Disk 3 fails, then can reconstruct: D2=D0⊕D1⊕D3⊕P0



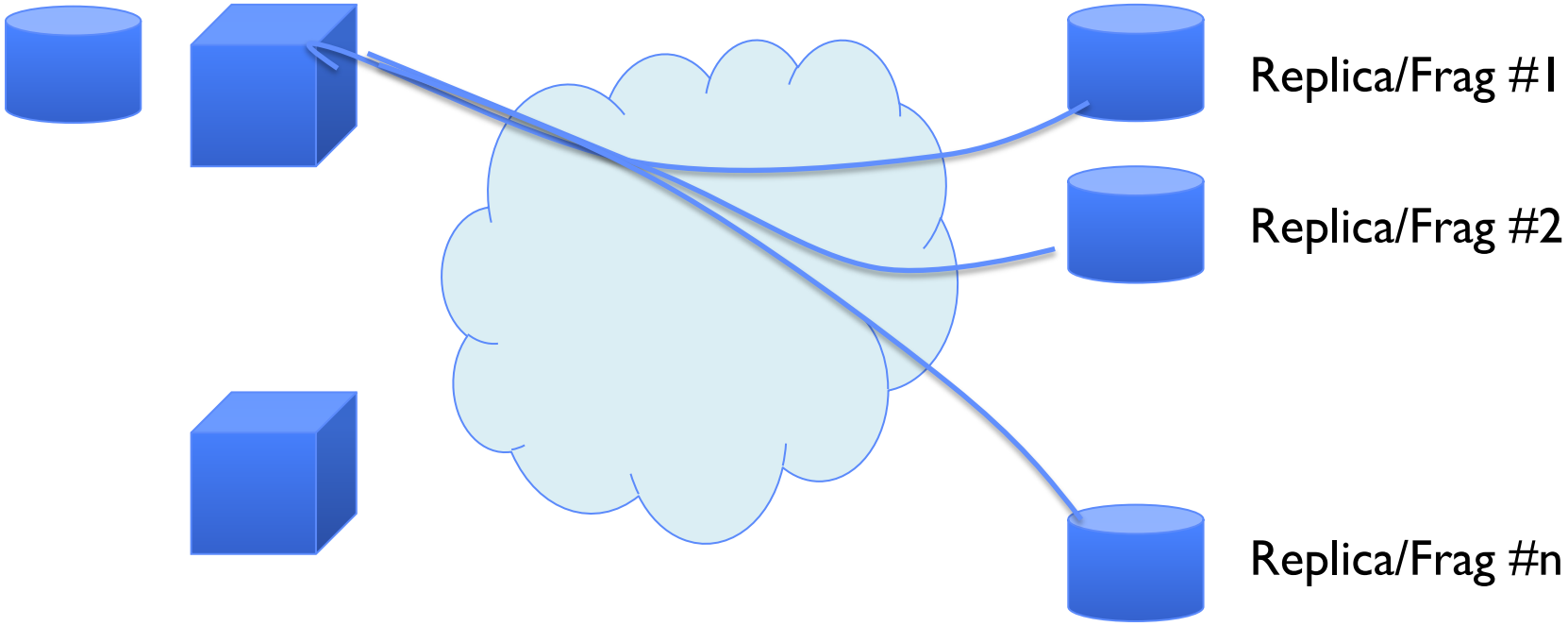
- Can spread information widely across internet for durability
  - RAID algorithms work over geographic scale

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
- More general option for general erasure code: Reed-Solomon codes
  - m data fragments
  - generate *n* - *m* extra fragments
  - can tolerate *n* – *m* failures
- Erasure codes not just for disk arrays. For example, geographic replication
  - E.g., split data into $m = 4$ fragments, generate $n = 16$ fragments and distribute across 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



HOW TO MAKE FILE SYSTEMS MORE *RELIABLE*?

File System Reliability: (Difference from Block-level reliability)

- What can happen if disk loses power or software crashes?
  - Some operations in progress may complete
  - Some operations in progress may be lost
  - Overwrite of a block may only partially complete

- Having RAID doesn't necessarily protect against all such failures
  - No protection against writing bad state
  - What if one disk of RAID group not written?
- File system needs durability (as a minimum!)
  - Data previously stored can be retrieved (maybe after some recovery step), regardless of failure

- But durability is not quite enough…!

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

Two Reliability Approaches

Careful Ordering and Recovery

- FAT & FFS + fsck
- Each step builds structure
- Data block ⇐ inode ⇐ free ⇐ directory
- Last step links it in to rest of FS
- Recover scans structure looking for incomplete actions

Versioning and Copy-on-Write

- ZFS, …
- Version files at some granularity
- Create new structure linking back to unchanged parts of old
- Last step is to declare that the new version is ready

File System Summary

- Deep interactions between mem management, file system, sharing
  - mmap(): map file or anonymous segment to memory
- Buffer Cache: Memory used to cache kernel resources, including disk blocks and name translations
  - Can contain “dirty” blocks (blocks yet on disk)
- File system operations involve multiple distinct updates to blocks on disk
  - Need to have all or nothing semantics
  - Crash may occur in the midst of the sequence
- Traditional file system perform check and recovery on boot
  - Along with careful ordering so partial operations result in loose fragments, rather than loss
- Copy-on-write provides richer function (versions) with much simpler recovery
  - Little performance impact since sequential write to storage device is nearly free