

CS162
Operating Systems and
Systems Programming
Lecture 23

Filesystems 3: Filesystem Case Studies (Con't),
Buffer Cache, Reliability

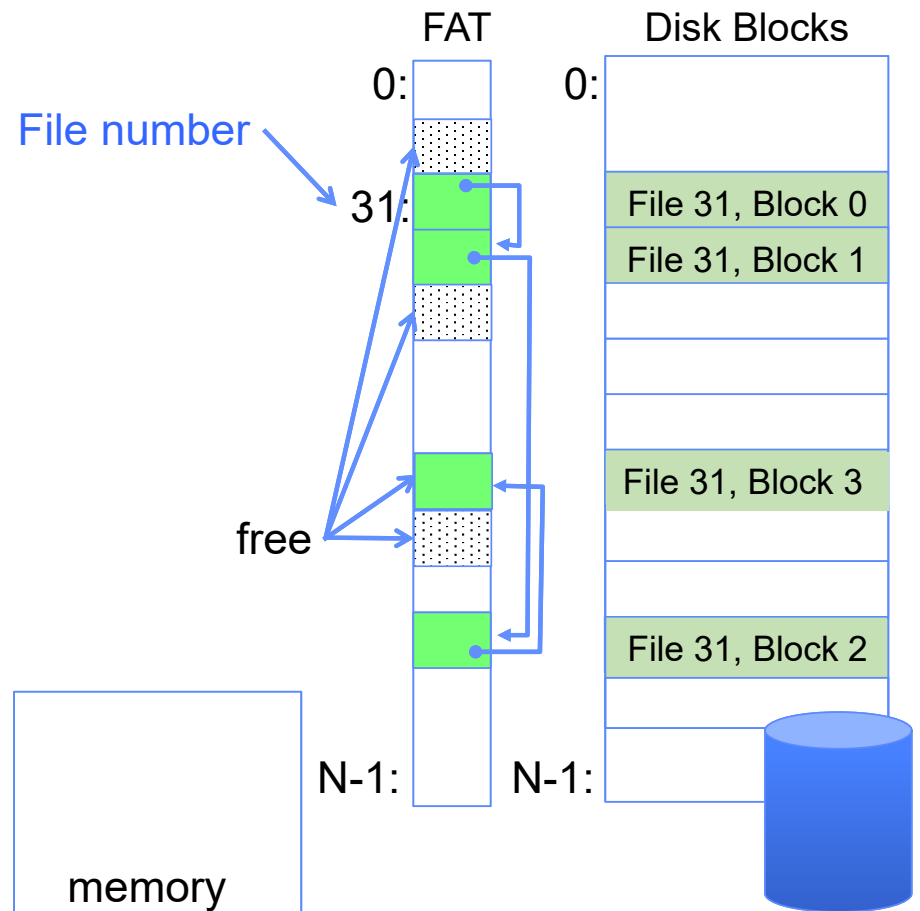
April 16th, 2024

Prof. John Kubiatowicz

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

Recall: FAT Properties

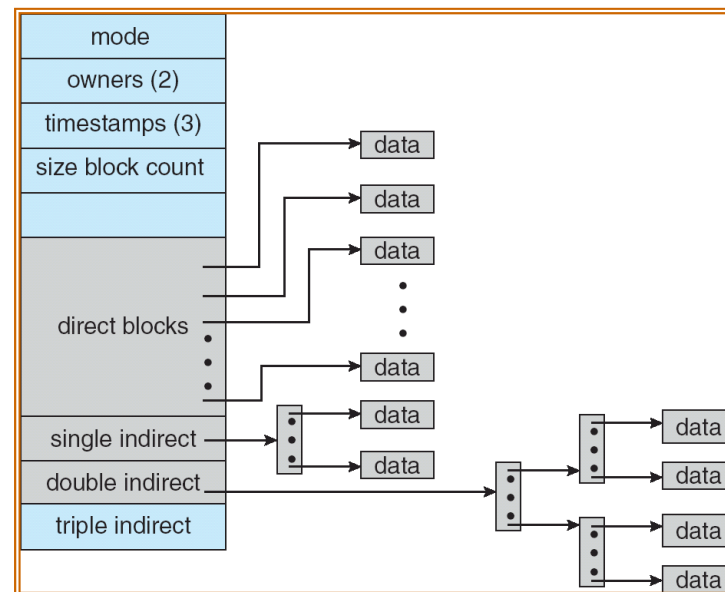
- File is collection of disk blocks (Microsoft calls them “clusters”)
- FAT is array of integers mapped 1-1 with disk blocks
 - Each integer is either:
 - » Pointer to next block in file; or
 - » End of file flag; or
 - » Free block flag
- File Number is index of root of block list for the file
 - Follow list to get block #
 - Directory must map name to block number at start of file
- But: Where is FAT stored?
 - Beginning of disk, before the data blocks
 - Usually 2 copies (to handle errors)



CASE STUDY: BERKELEY FAST FILE SYSTEM (FFS)

Recall: Multilevel Indexed Files (Original 4.1 BSD)

- Sample file in multilevel indexed format:
 - 10 direct ptrs, 1K blocks
 - 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



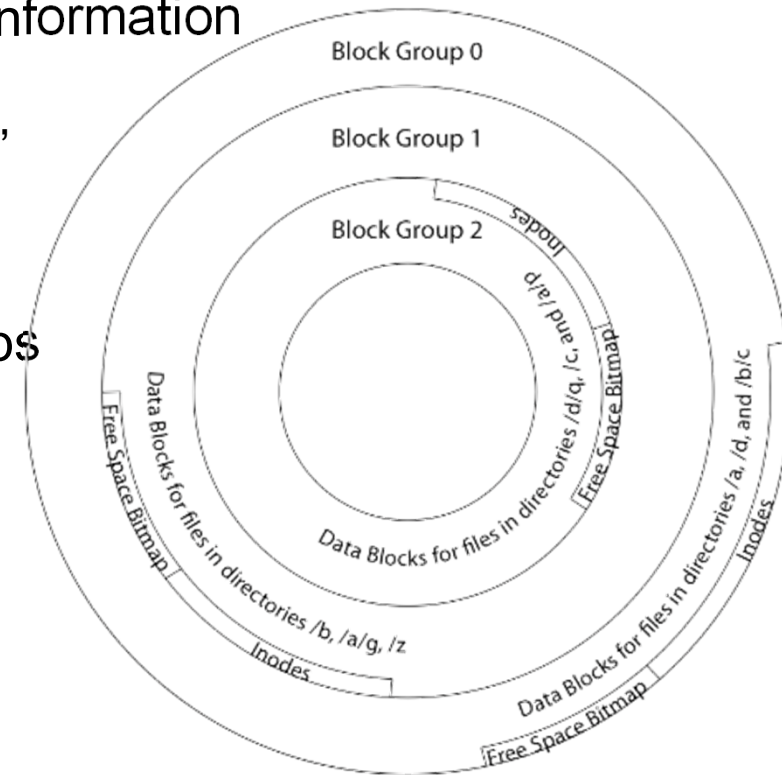
- UNIX 4.1 Pros and cons
 - Pros: Simple (more or less)
Files can easily expand (up to a point)
Small files particularly cheap and easy
 - Cons: Lots of seeks
Very large files must read many indirect block (four I/Os per block!)

Recall: FFS Changes in Inode Placement: Motivation

- In early UNIX and DOS/Windows' FAT file system, headers stored in special array in outermost cylinders
 - Fixed size, set when disk is formatted
 - » At formatting time, a fixed number of inodes are created
 - » Each is given a unique number, called an “inumber”
- Problem #1: Inodes all in one place (outer tracks)
 - Head crash potentially destroys all files by destroying inodes
 - Inodes not close to the data that they point to
 - » To read a small file, seek to get header, seek back to data
- Problem #2: When create a file, don't know how big it will become (in UNIX, most writes are by appending)
 - How much contiguous space do you allocate for a file?
 - Makes it hard to optimize for performance

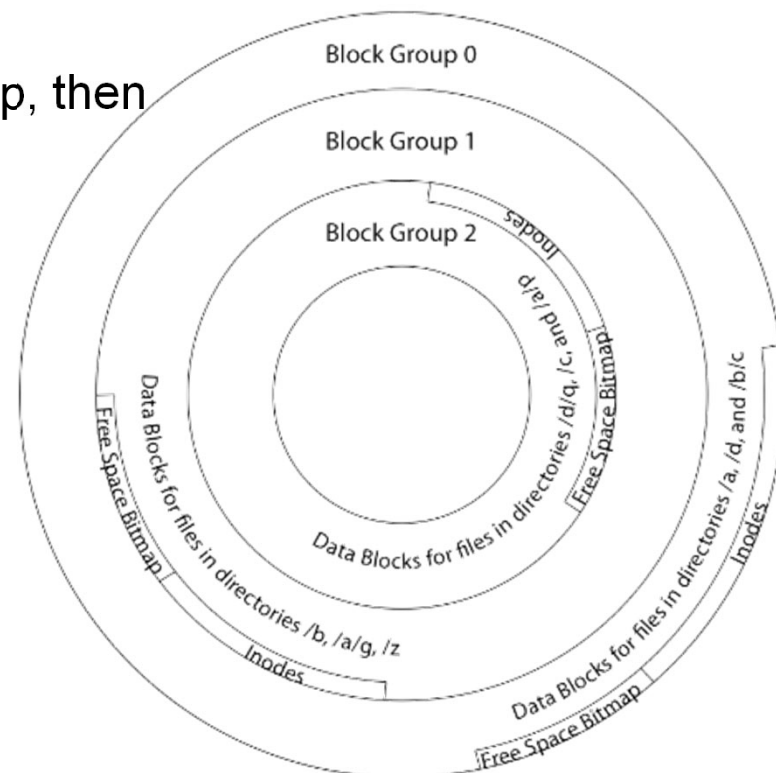
FFS Locality: Block Groups

- 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

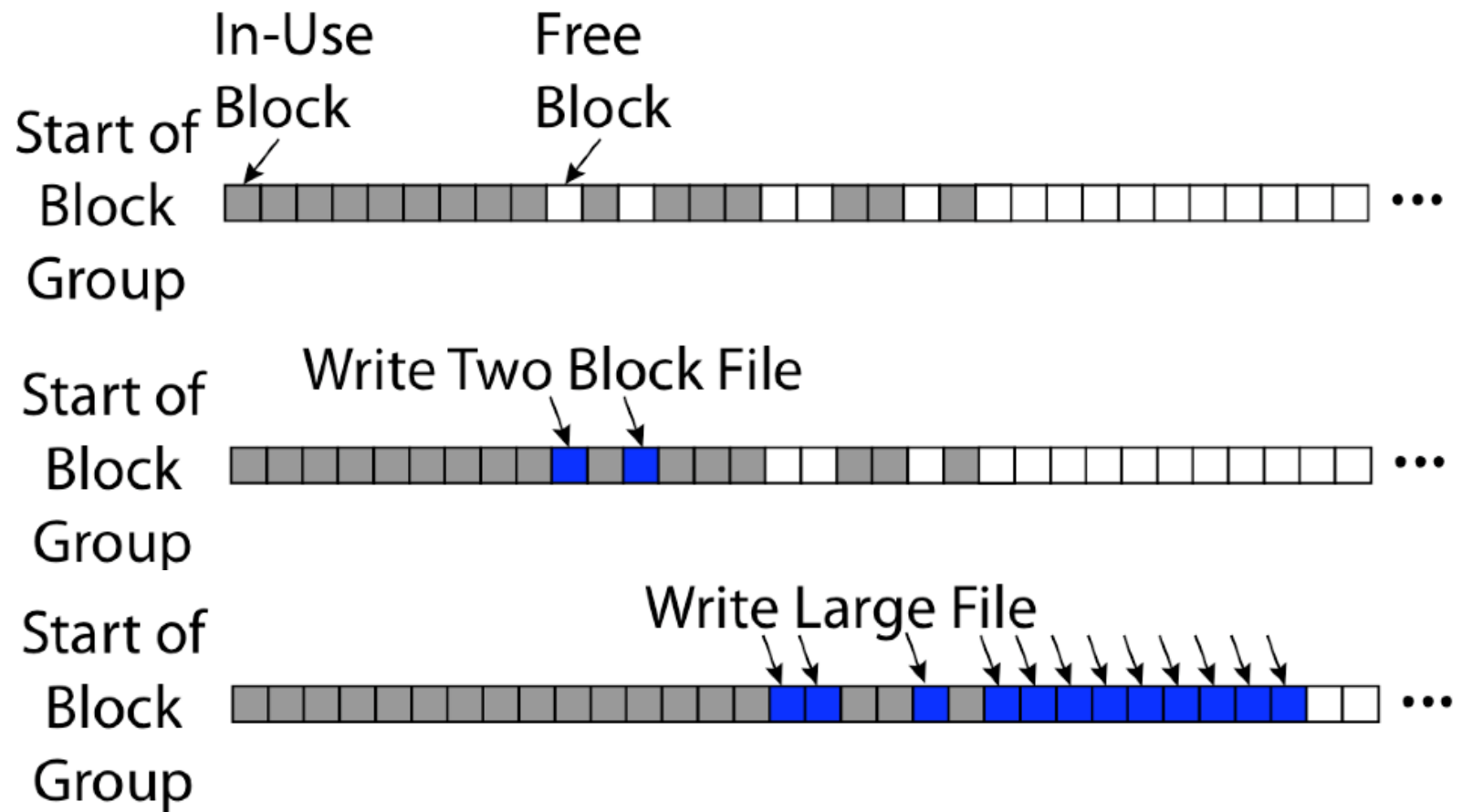


FFS Locality: Block Groups (Con't)

- First-Free allocation of new file blocks
 - To expand file, first try successive blocks in bitmap, then choose new range of blocks
 - Few little holes at start, big sequential runs at end of group
 - Avoids fragmentation
 - Sequential layout for big files
- **Important: keep 10% or more free!**
 - **Reserve space in the Block Group**
- Summary: FFS Inode Layout Pros
 - For small directories, can fit all data, file headers, etc. in same cylinder \Rightarrow no seeks!
 - File headers much smaller than whole block (a few hundred bytes), so multiple headers fetched from disk at same time
 - Reliability: whatever happens to the disk, you can find many of the files (even if directories disconnected)

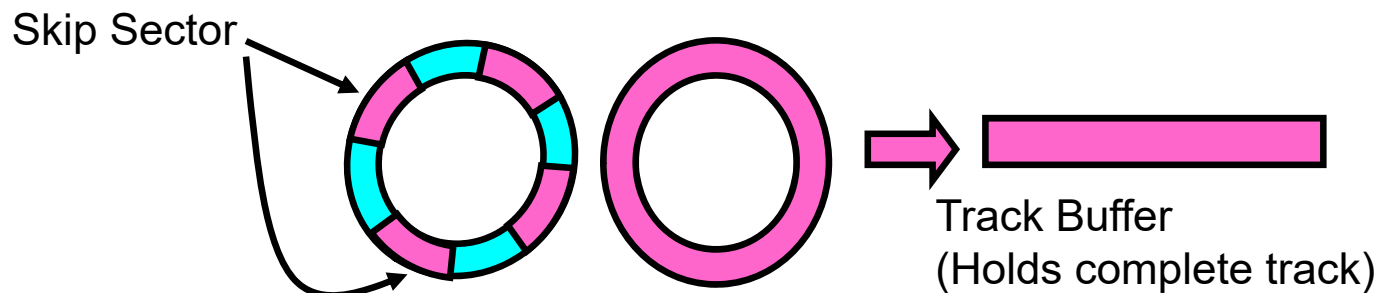


UNIX 4.2 BSD FFS First Fit Block Allocation



Attack of the Rotational Delay

- Problem 3: Missing blocks due to rotational delay
 - Issue: Read one block, do processing, and read next block. In meantime, disk has continued turning: missed next block! Need 1 revolution/block!



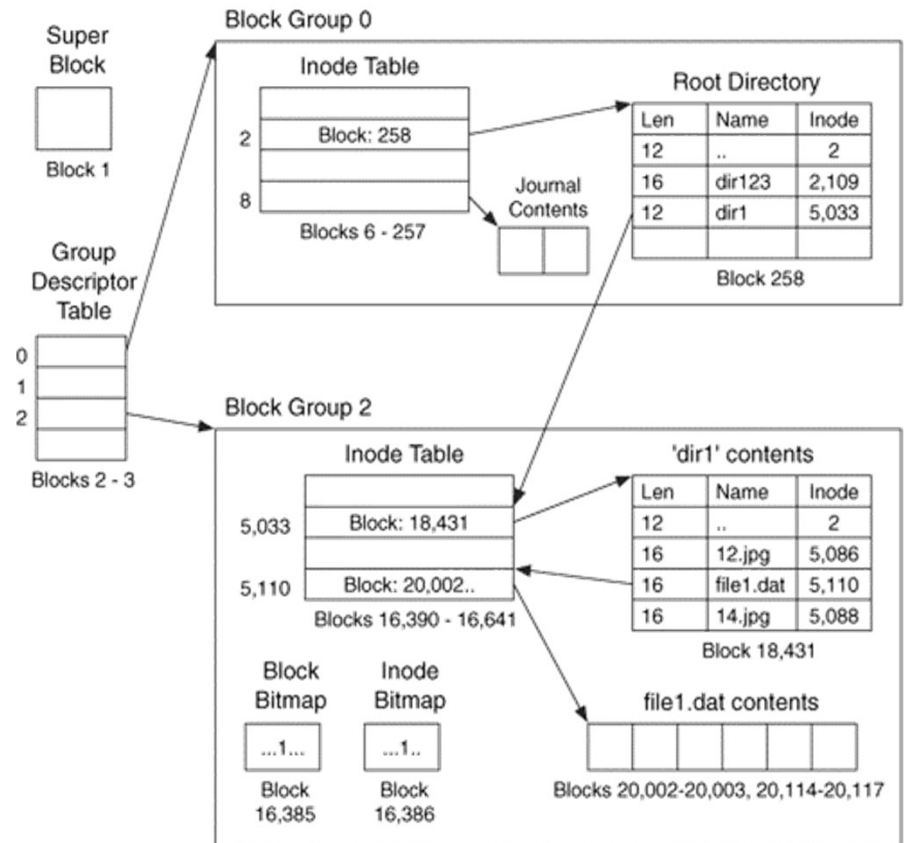
- Solution 1: Skip sector positioning (“interleaving”)
 - » Place the blocks from one file on every other block of a track: give time for processing to overlap rotation
 - » Can be done by OS or in modern drives by the disk controller
- Solution 2: Read ahead: read next block right after first, even if application hasn’t asked for it yet
 - » This can be done either by OS (read ahead)
 - » By disk itself (track buffers) - many disk controllers have internal RAM that allows them to read a complete track
- Modern disks + controllers do many things “under the covers”
 - Track buffers, elevator algorithms, bad block filtering

UNIX 4.2 BSD FFS

- Pros
 - Efficient storage for both small and large files
 - Locality for both small and large files
 - Locality for metadata and data
 - No defragmentation necessary!
- Cons
 - Inefficient for tiny files (a 1 byte file requires both an inode and a data block)
 - Inefficient encoding when file is mostly contiguous on disk
 - Need to reserve 10-20% of free space to prevent fragmentation

Linux Example: Ext2/3 Disk Layout

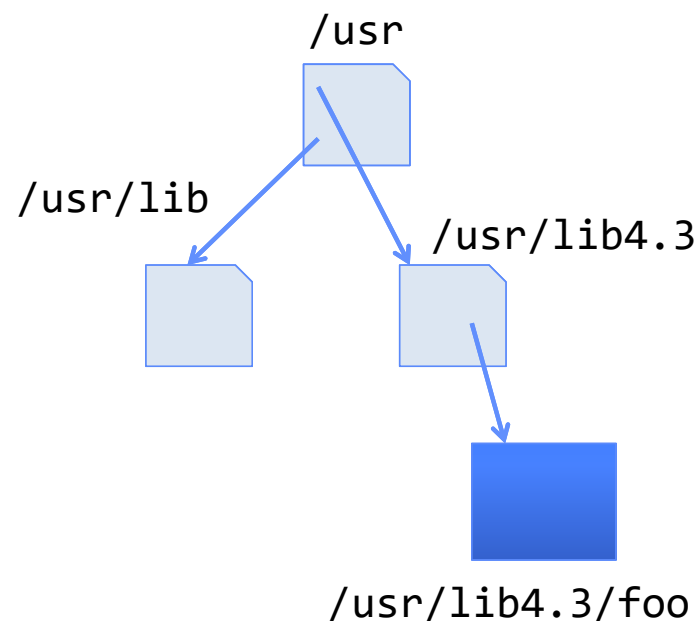
- Disk divided into block groups
 - Provides locality
 - Each group has two block-sized bitmaps (free blocks/inodes)
 - Block sizes settable at format time: 1K, 2K, 4K, 8K...
- Actual inode structure similar to 4.2 BSD
 - with 12 direct pointers
- Ext3: Ext2 with Journaling
 - Several degrees of protection with comparable overhead
 - We will talk about Journalling later



- Example: create a file1.dat under /dir1/ in Ext3

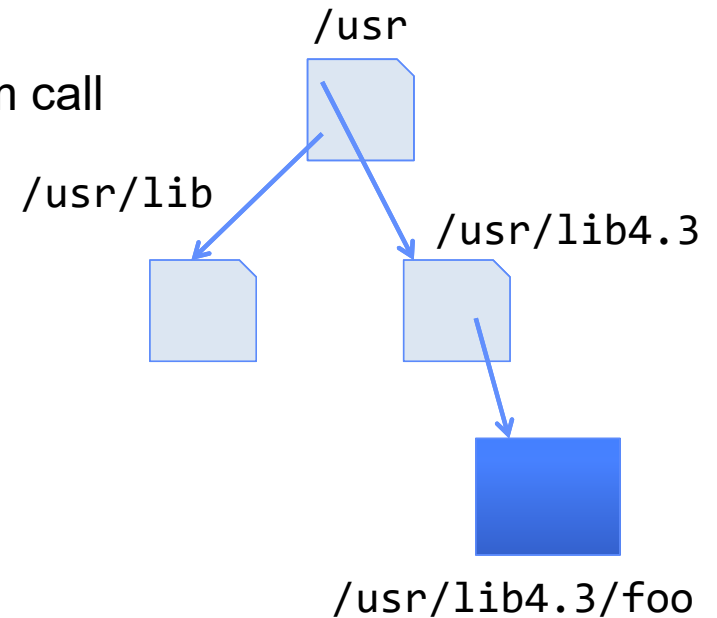
Recall: Directory Abstraction

- Directories are specialized files
 - Contents: **List of pairs <file name, file number>**
- System calls to access directories
 - open / creat traverse the structure
 - mkdir /rmdir add/remove entries
 - link / unlink (rm)
- libc support
 - `DIR * opendir (const char *dirname)`
 - `struct dirent * readdir (DIR *dirstream)`
 - `int readdir_r (DIR *dirstream, struct dirent *entry, struct dirent **result)`



Hard Links

- Hard link
 - Mapping from name to file number in the directory structure
 - First hard link to a file is made when file created
 - Create extra hard links to a file with the `link()` system call
 - Remove links with `unlink()` system call
- When can file contents be deleted?
 - When there are no more hard links to the file
 - Inode maintains reference count for this purpose

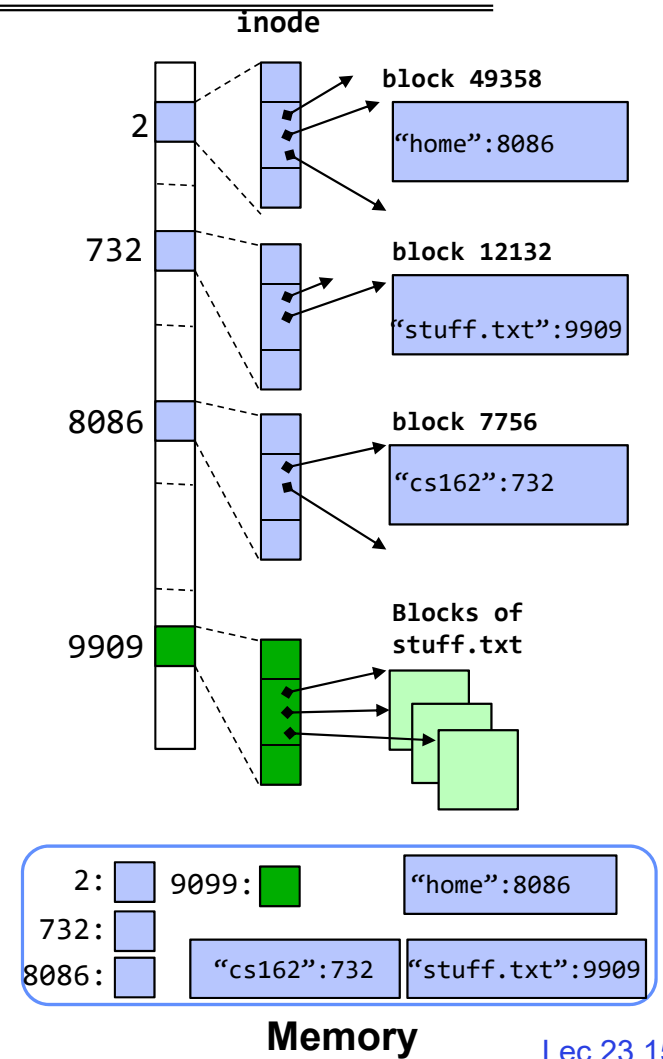


Soft Links (Symbolic Links)

- Soft link or Symbolic Link or Shortcut
 - Directory entry contains the path and name of the file
 - Map one name to another name
- Contrast these two different types of directory entries:
 - Normal directory entry: <file name, **file #**>
 - Symbolic link: <file name, **dest. file name**>
- OS looks up destination file name **each time** program accesses source file name
 - Lookup can fail (error result from **open**)
- Unix: Create soft links with **symlink** syscall

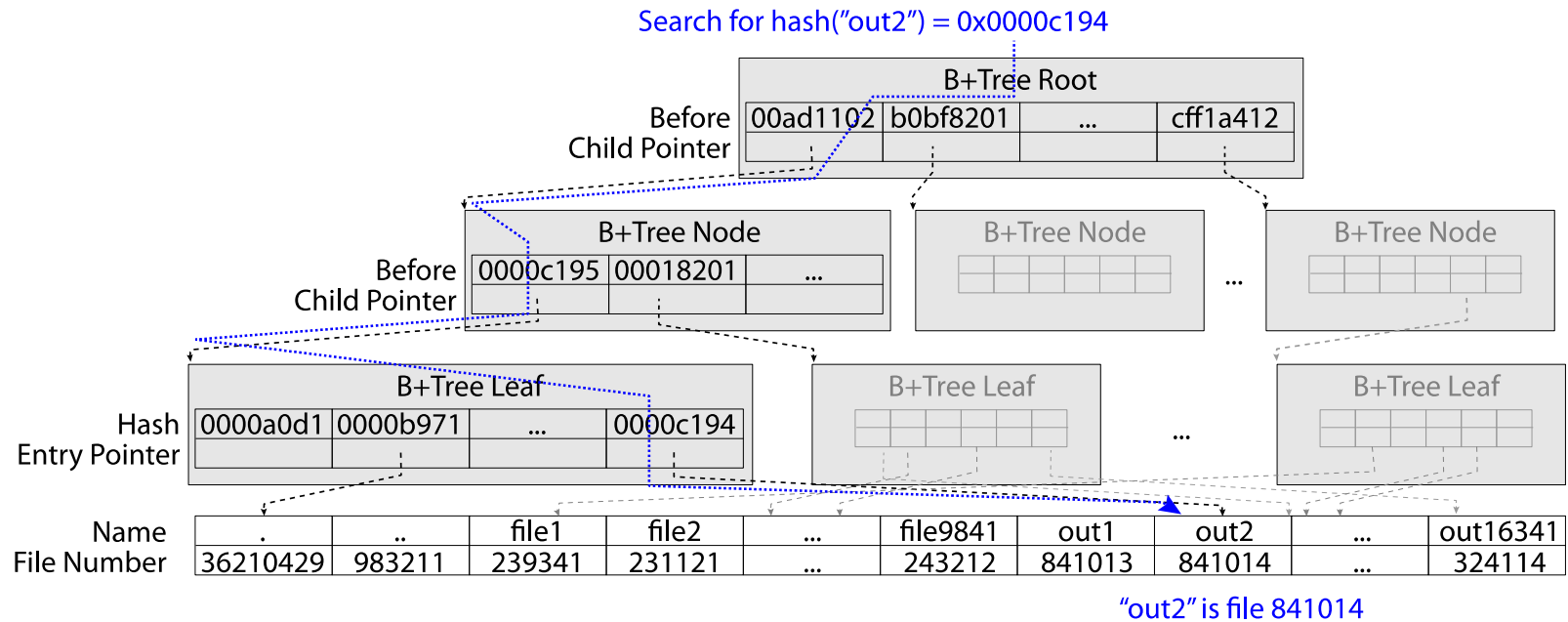
Directory Traversal

- What happens when we open `/home/cs162/stuff.txt`?
- “/” - inumber for root inode configured into kernel, say 2
 - Read inode 2 from its position in inode array on disk
 - Extract the direct and indirect block pointers
 - Determine block that holds root directory (say block 49358)
 - Read that block, scan it for “home” to get inumber for this directory (say 8086)
- Read inode 8086 for `/home`, extract its blocks, read block (say 7756), scan it for “cs162” to get its inumber (say 732)
- Read inode 732 for `/home/cs162`, extract its blocks, read block (say 12132), scan it for “stuff.txt” to get its inumber, say 9909
- Read inode 9909 for `/home/cs162/stuff.txt`
- Set up file description to refer to this inode so reads / write can access the data blocks referenced by its direct and indirect pointers
- **Check permissions on the final inode and each directory’s inode...**



Large Directories: B-Trees (dirhash)

in FreeBSD, NetBSD, OpenBSD



Administrivia

- Homework 5: Due April 23rd
- Project 3: Design reviews this week
- Midterm 3: April 25th
 - Everything fair game with focus on last 1/3 of class
 - Three *hand-written* cheat-sheets, double sided
- Class attendance: No credit for people who use the same photo!
- Last chance to suggest topics for final lecture!



<https://tinyurl.com/25ep5ep3>

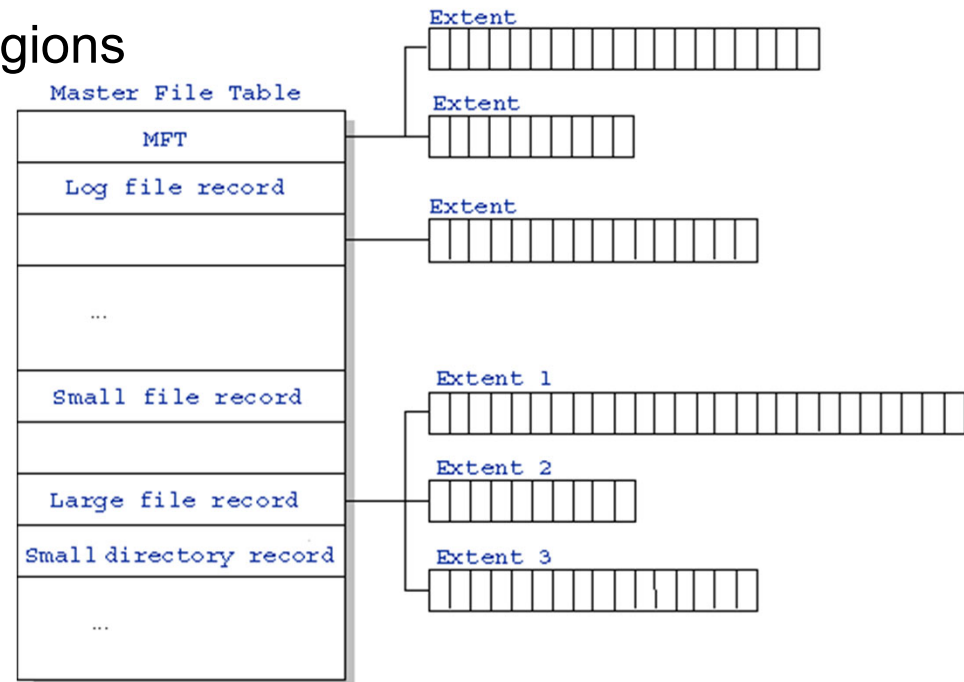
CASE STUDY: WINDOWS NTFS

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

NTFS

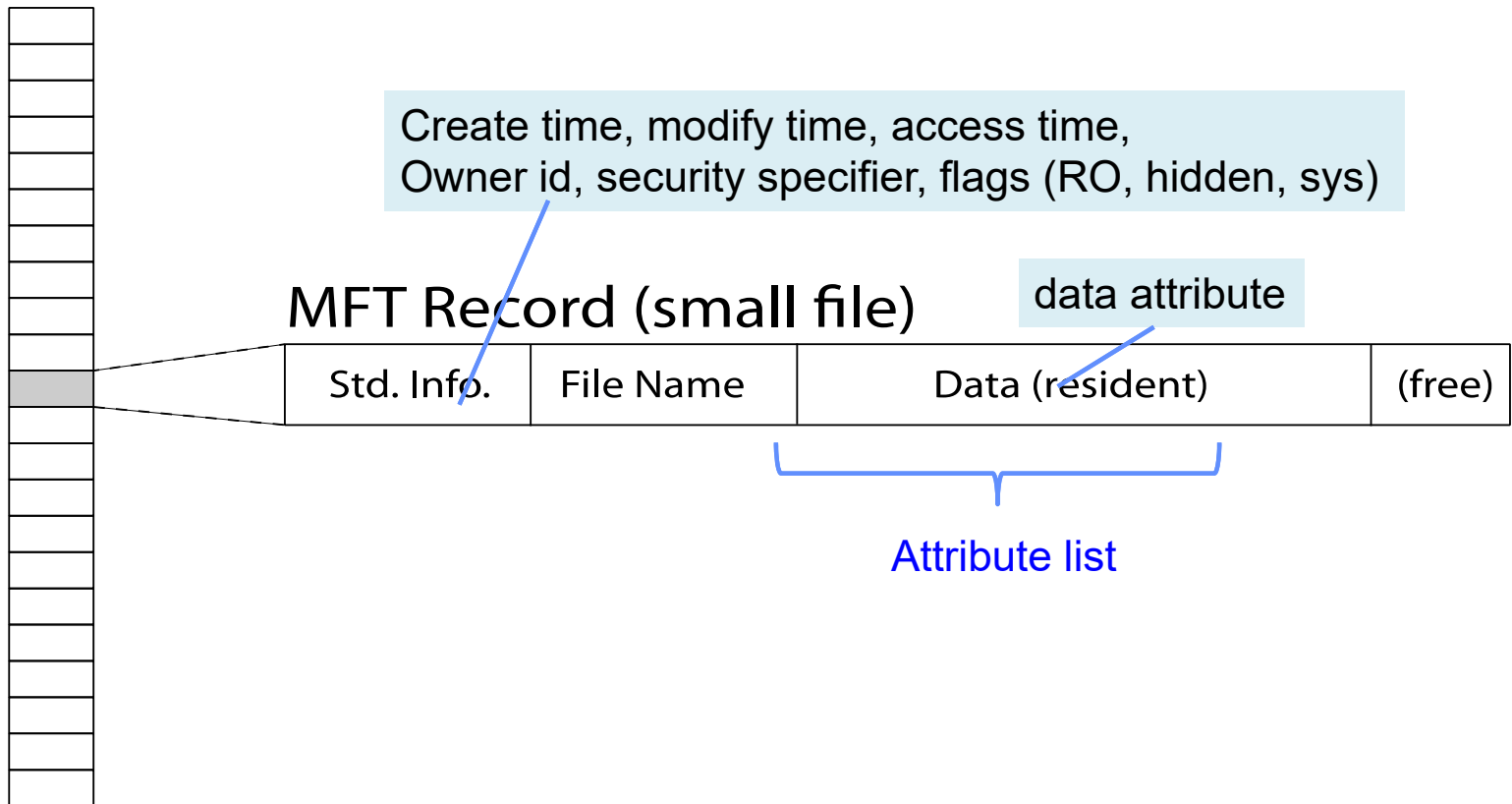
- Master File Table
 - Database with Flexible 1KB entries for metadata/data
 - Variable-sized attribute records (data or metadata)
 - Extend with variable depth tree (non-resident)
- Extents – variable length contiguous regions
 - Block pointers cover runs of blocks
 - Similar approach in Linux (ext4)
 - File create can provide hint as to size of file
- Journaling for reliability
 - Discussed later



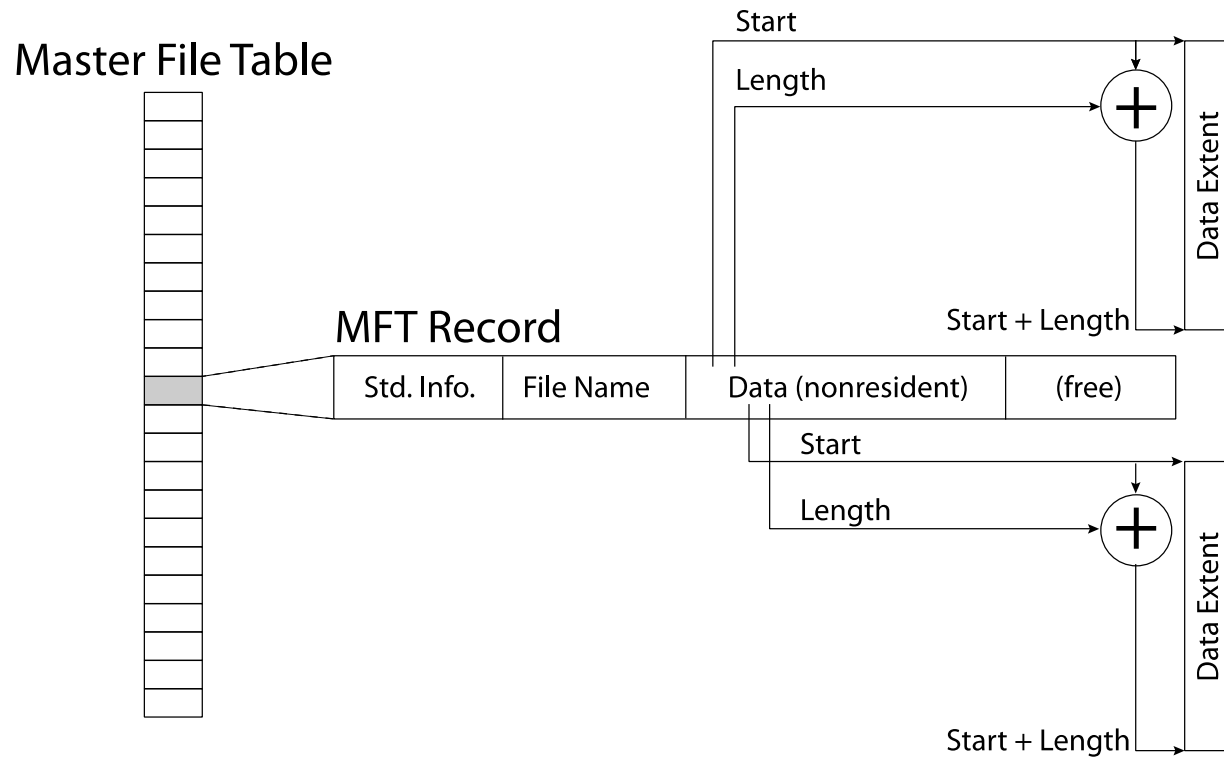
<http://ntfs.com/ntfs-mft.htm>

NTFS Small File: Data stored with Metadata

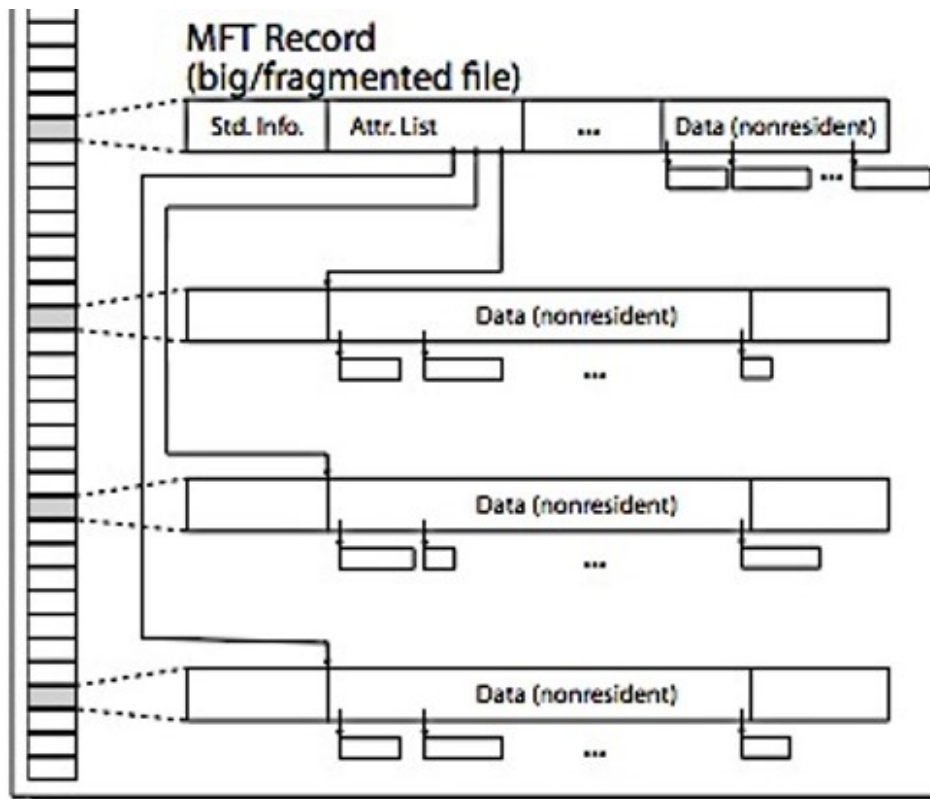
Master File Table



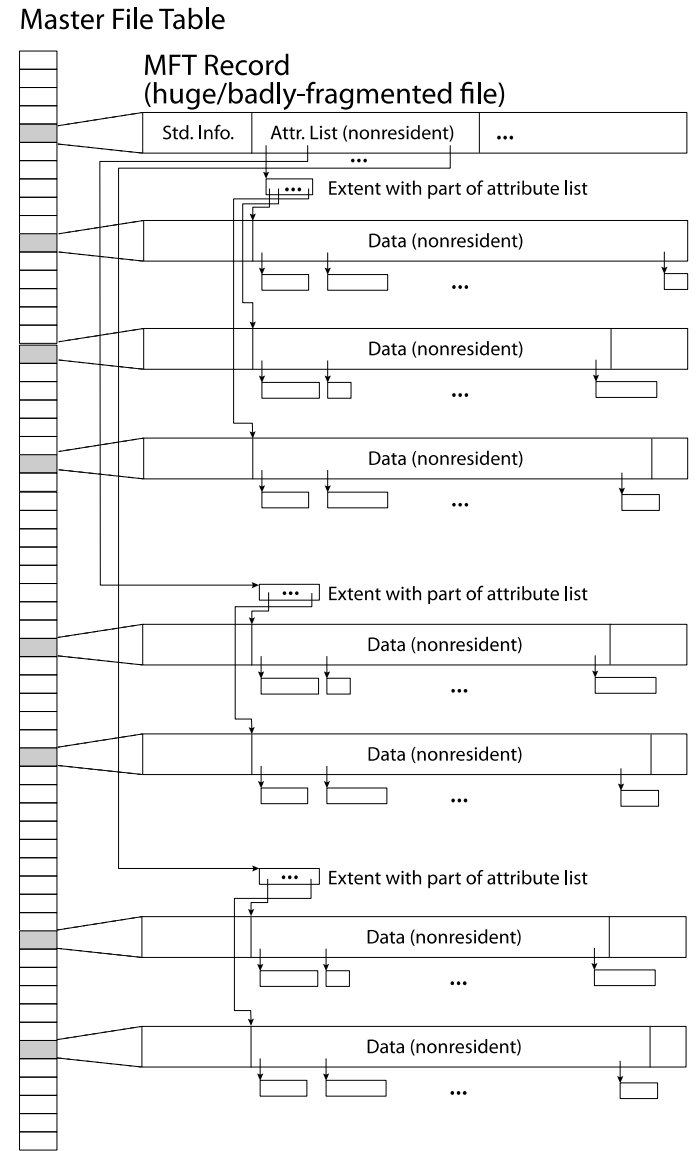
NTFS Medium File: Extents for File Data



NTFS Large File: Pointers to Other MFT Records



NTFS Huge, Fragmented File: Many MFT Records



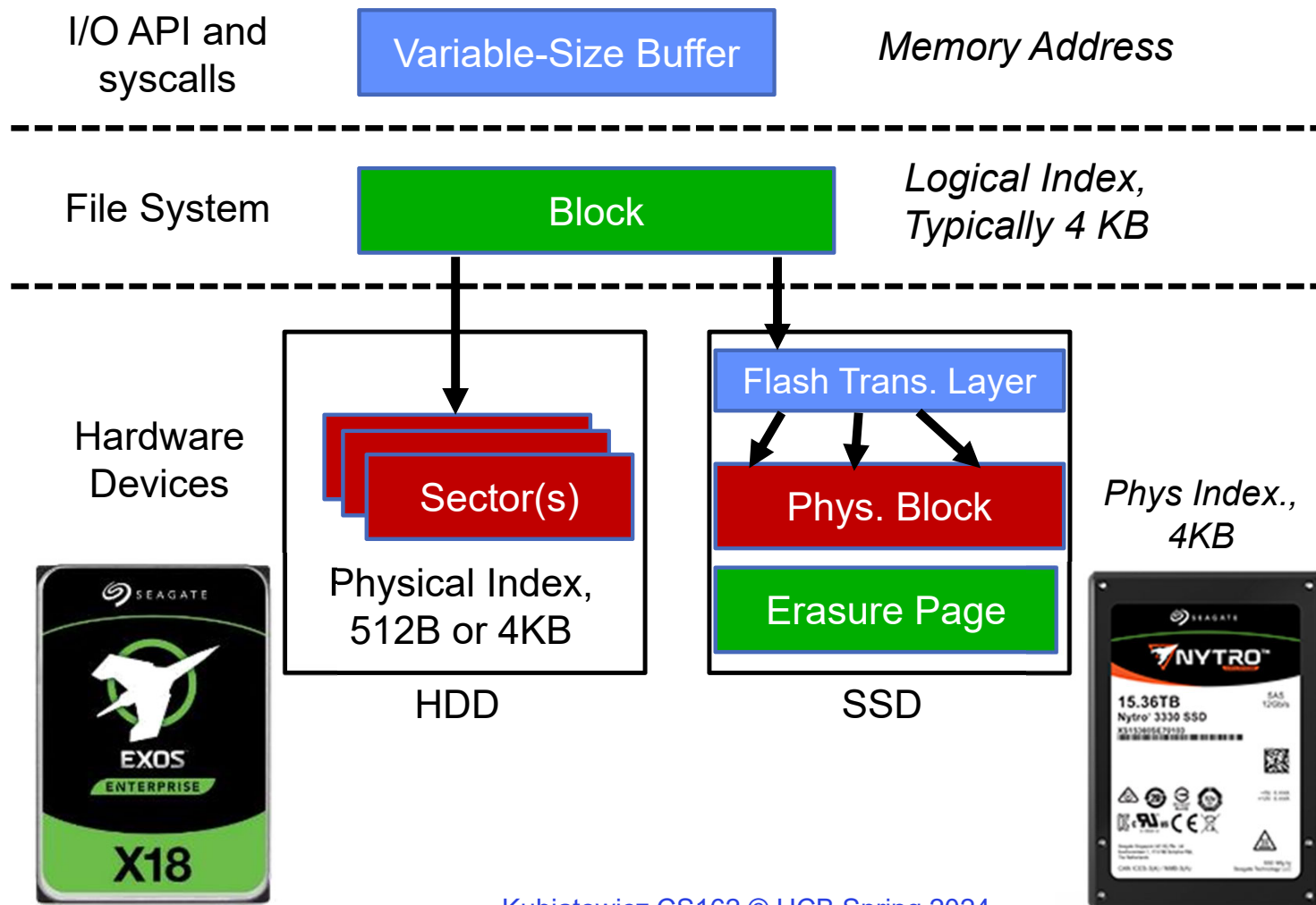
NTFS Directories

- Directories implemented as B Trees
- File's number identifies its entry in MFT
- MFT entry always has a file name attribute
 - Human readable name, file number of parent dir
- Hard link? Multiple file name attributes in MFT entry

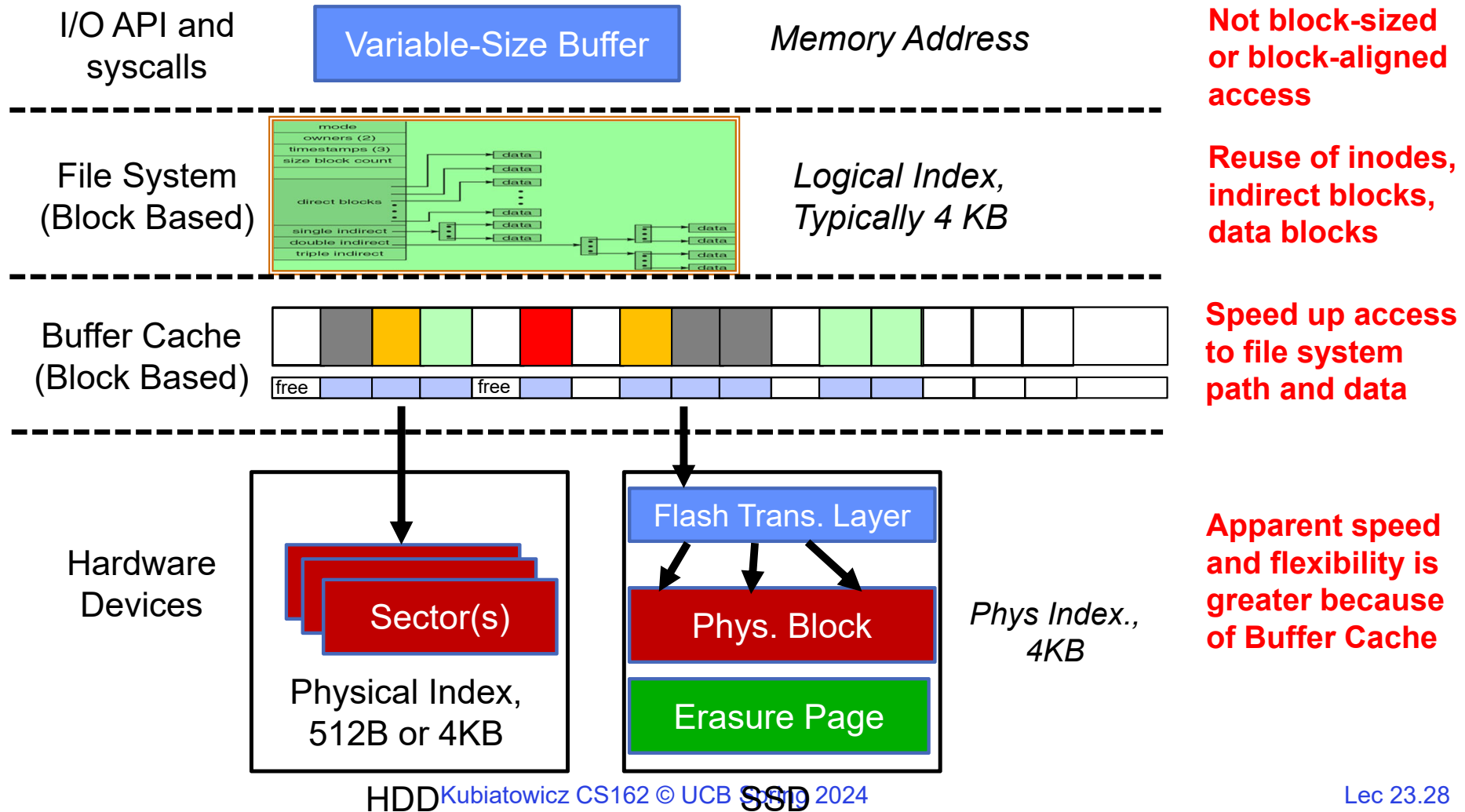


THE BUFFER CACHE

Recall: From Storage to File Systems

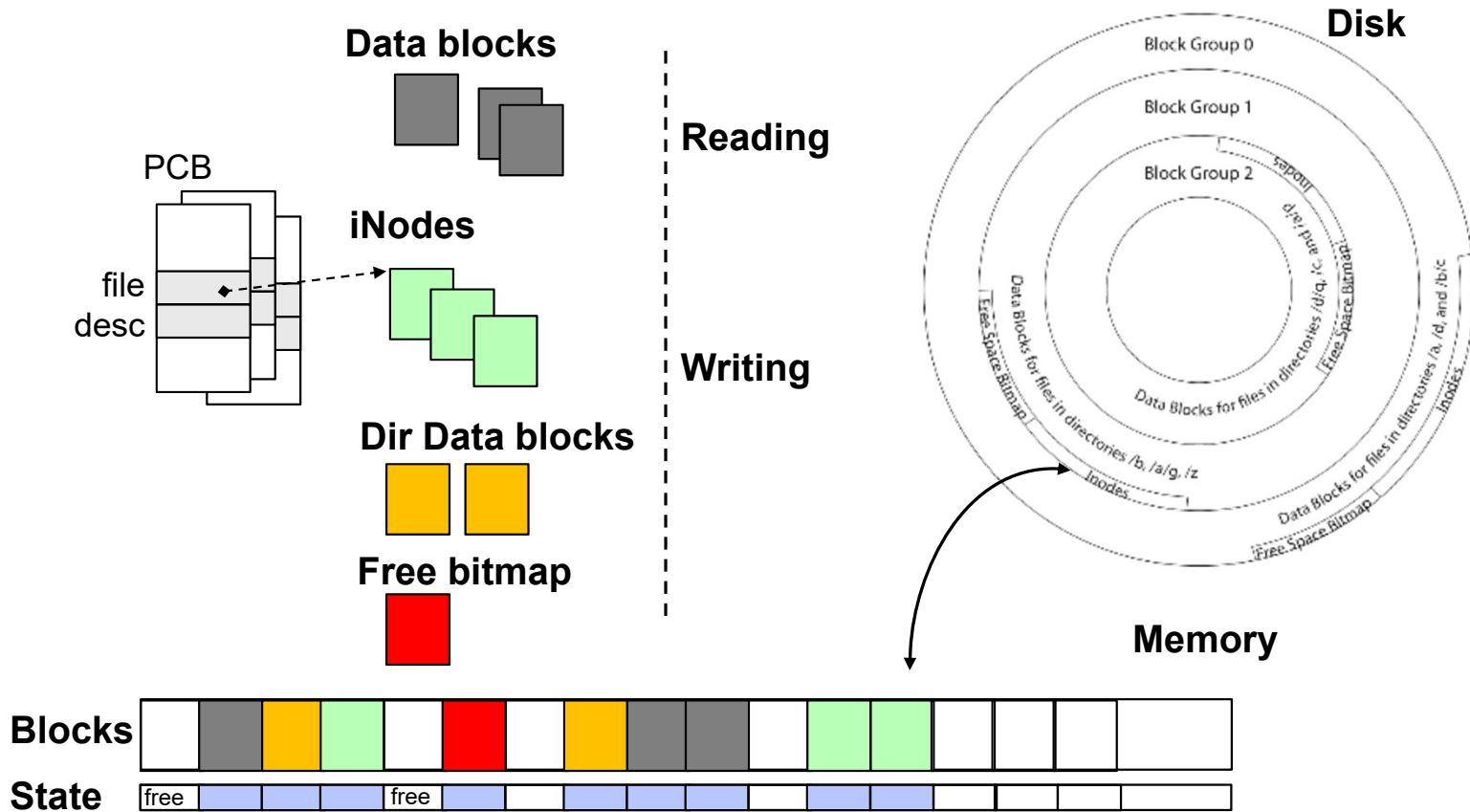


Need for Cache Between FileSystem and Devices



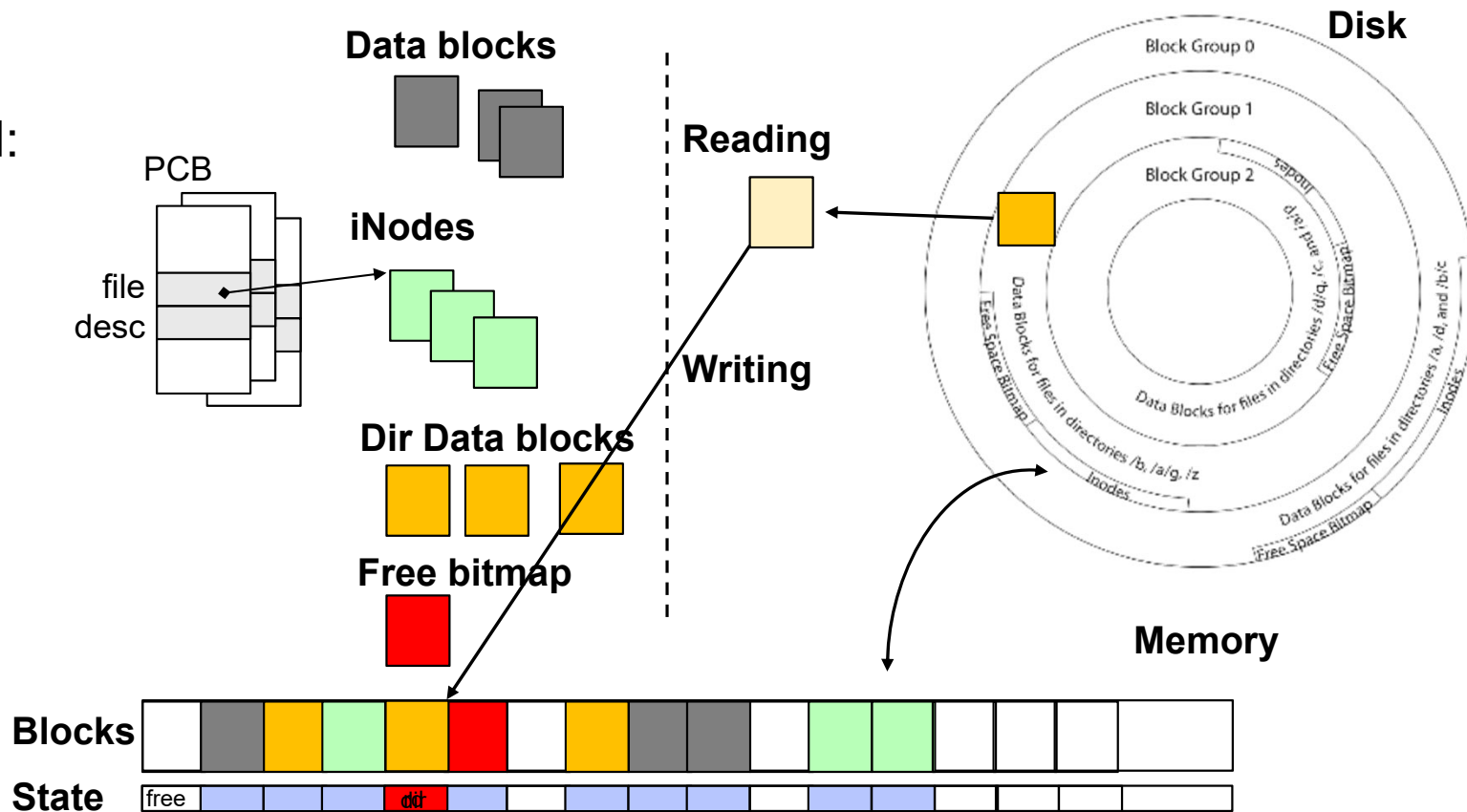
File System Buffer Cache

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



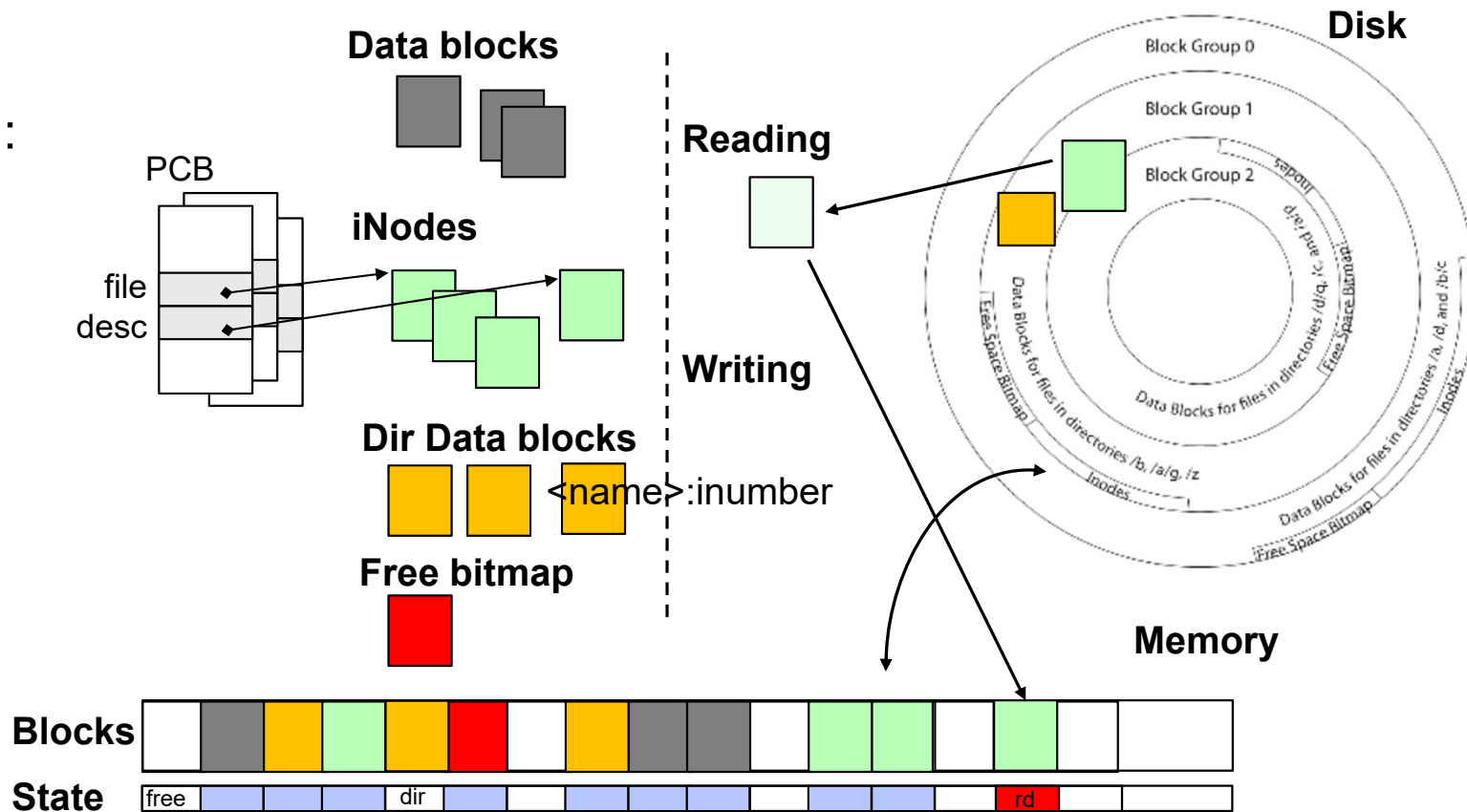
File System Buffer Cache: open

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



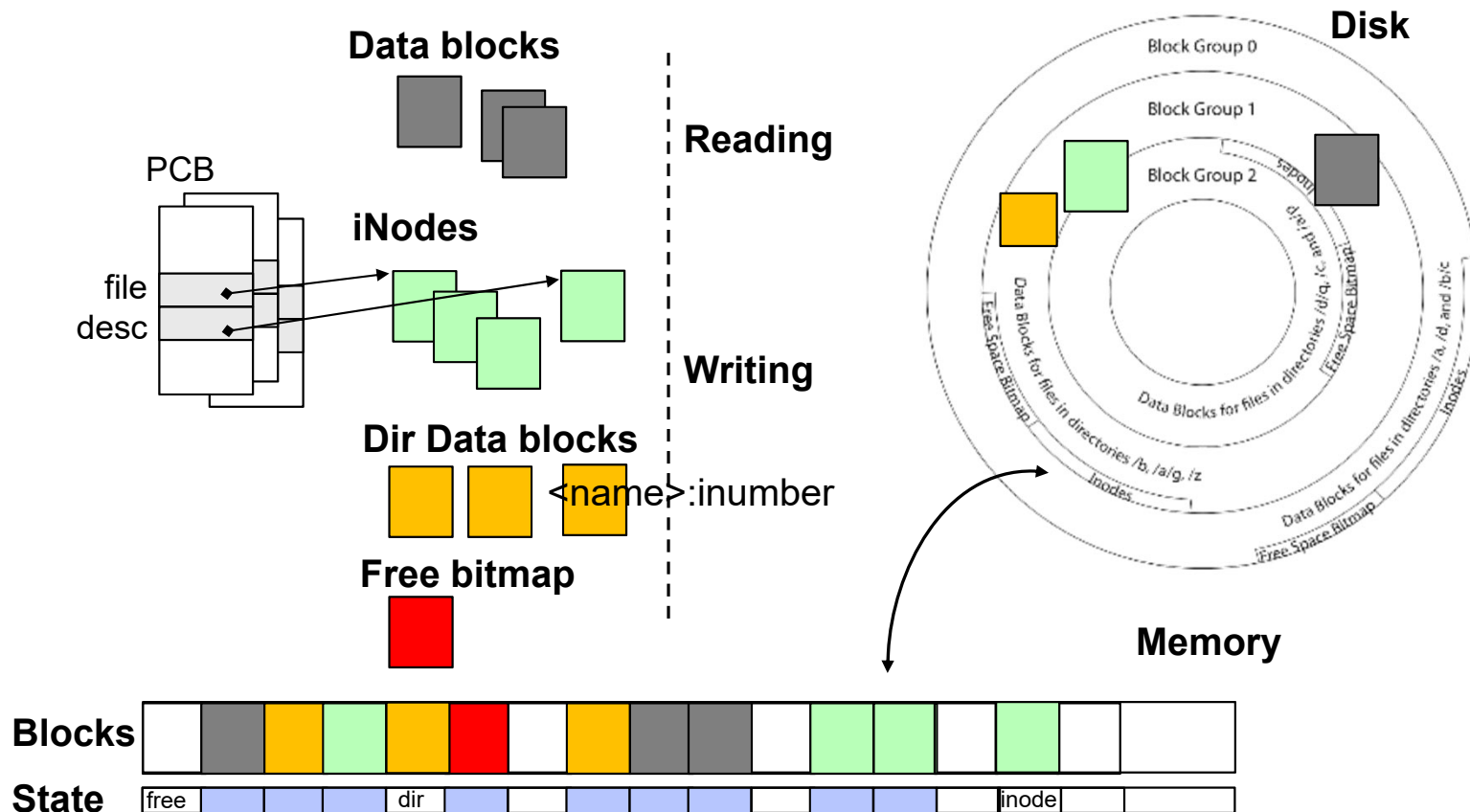
File System Buffer Cache: open

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



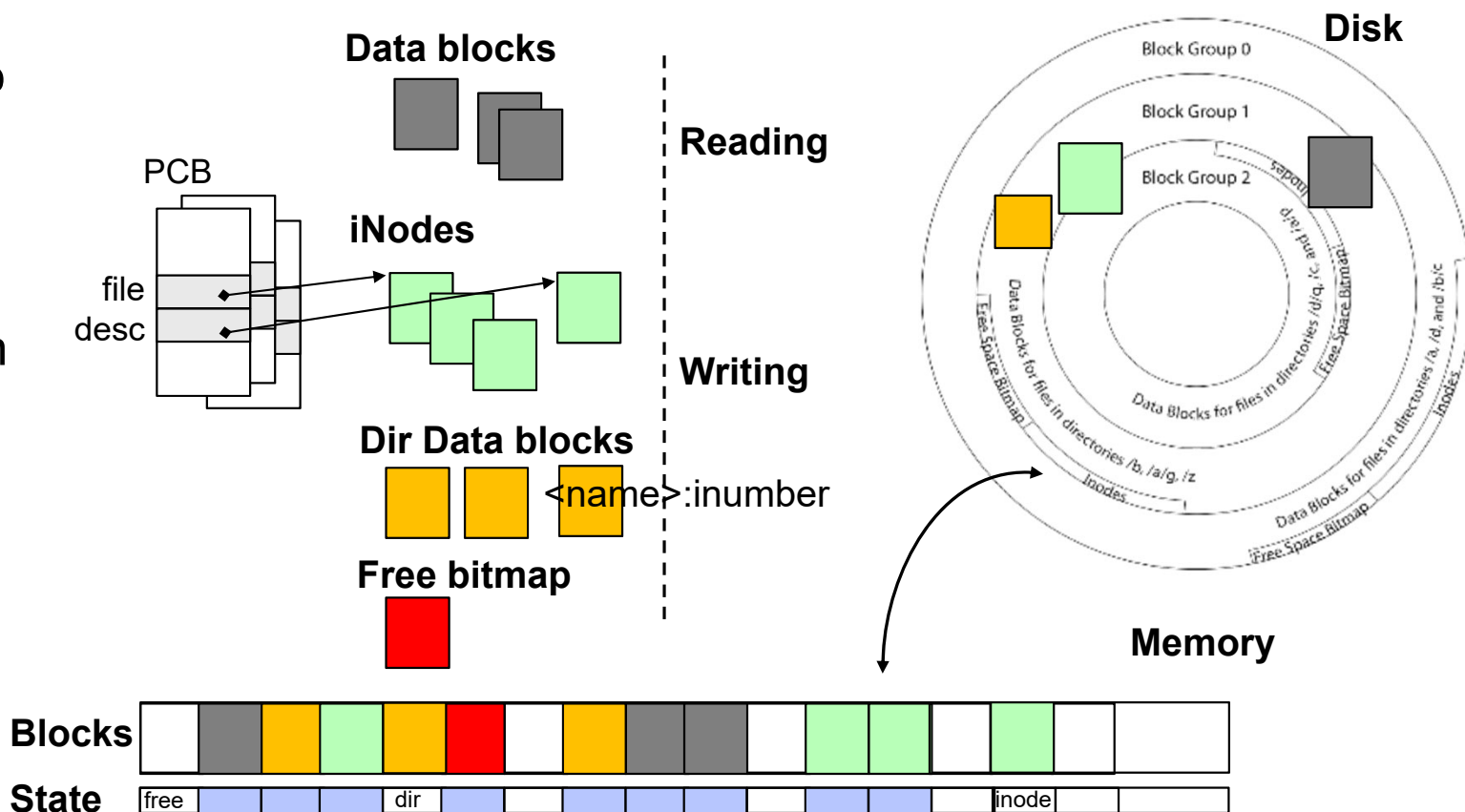
File System Buffer Cache: Read?

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



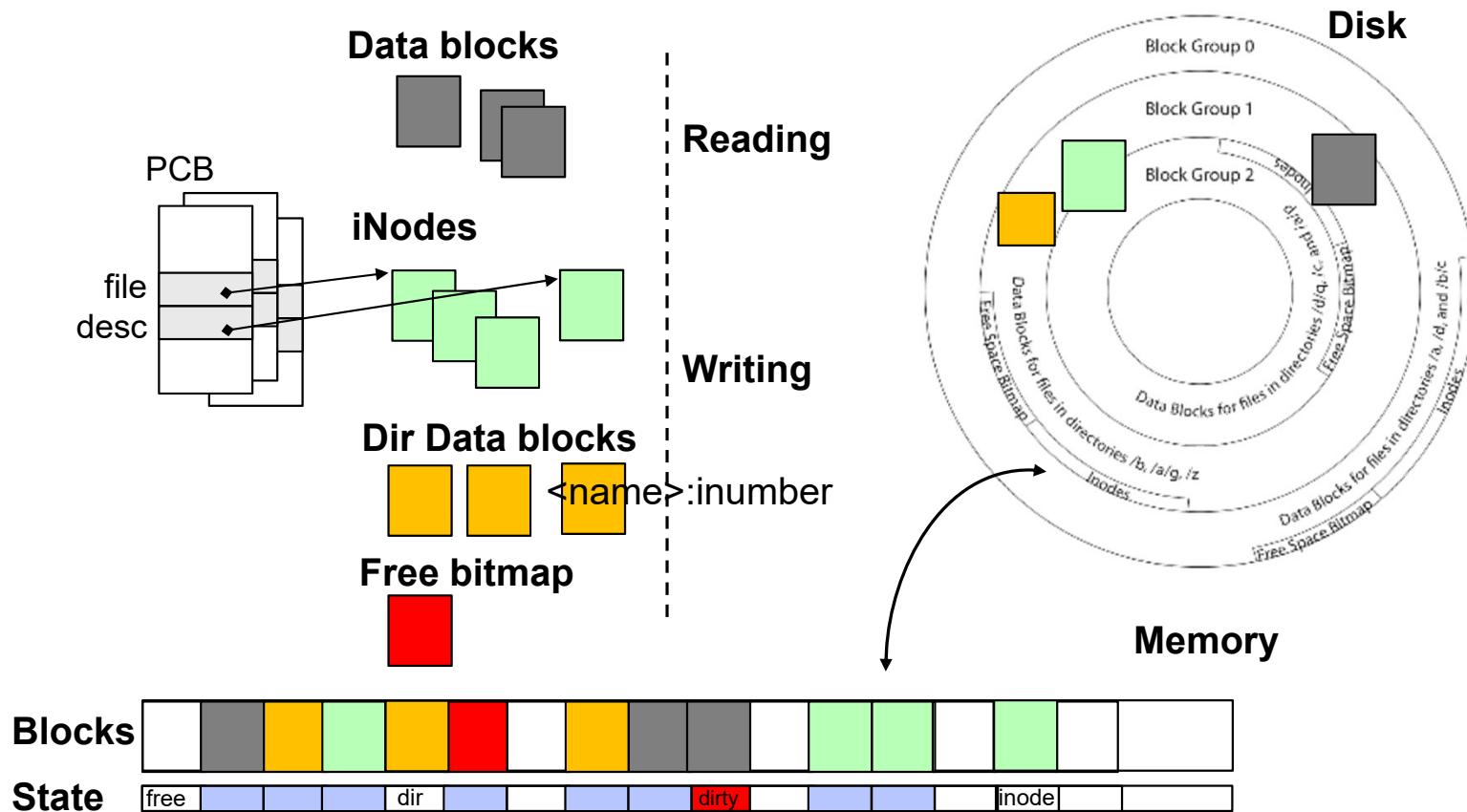
File System Buffer Cache: Write?

- 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 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 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** 😞

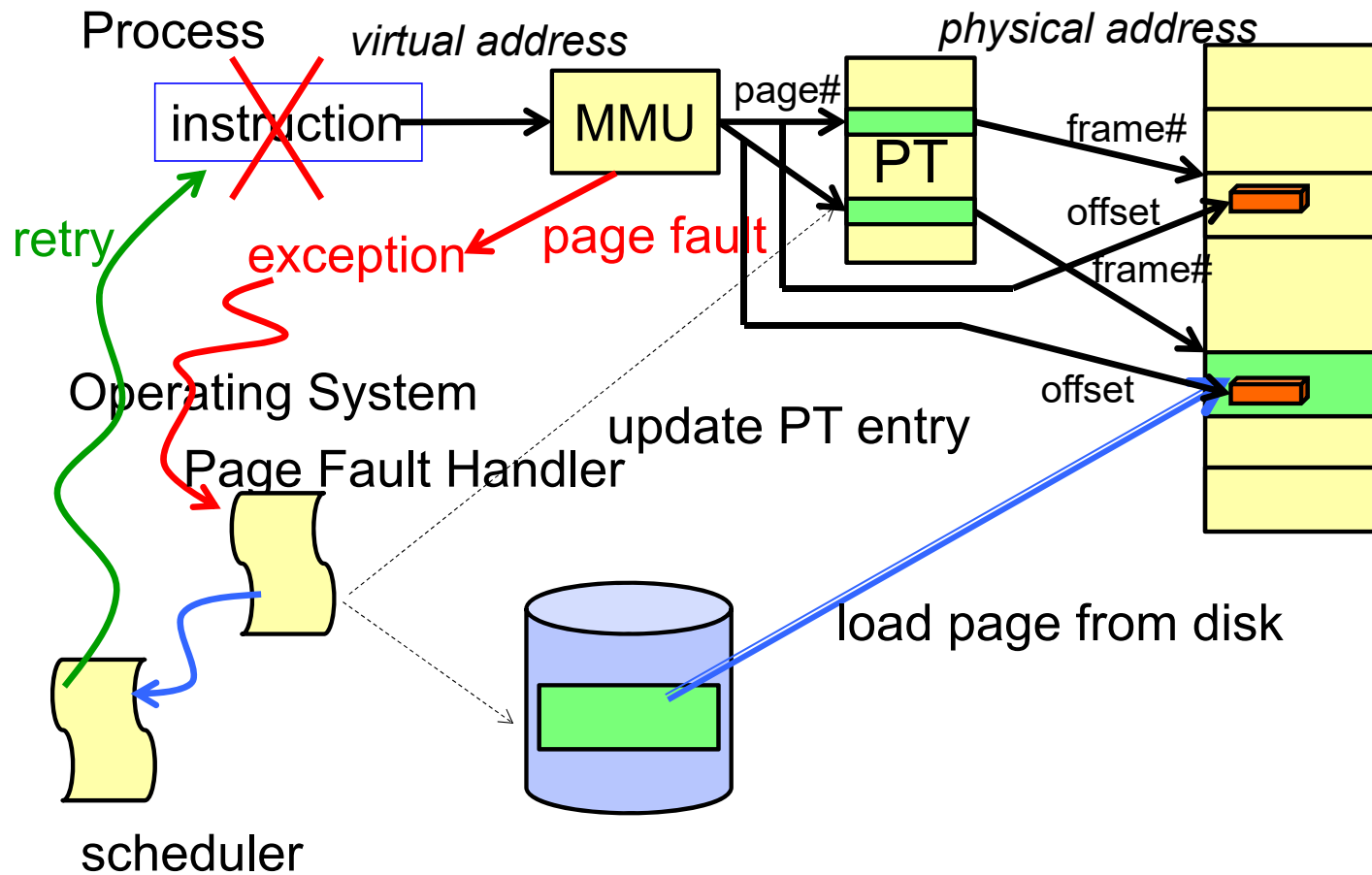
Takeaway: File systems need recovery mechanisms

QUICK ASIDE: MEMORY MAPPED FILES

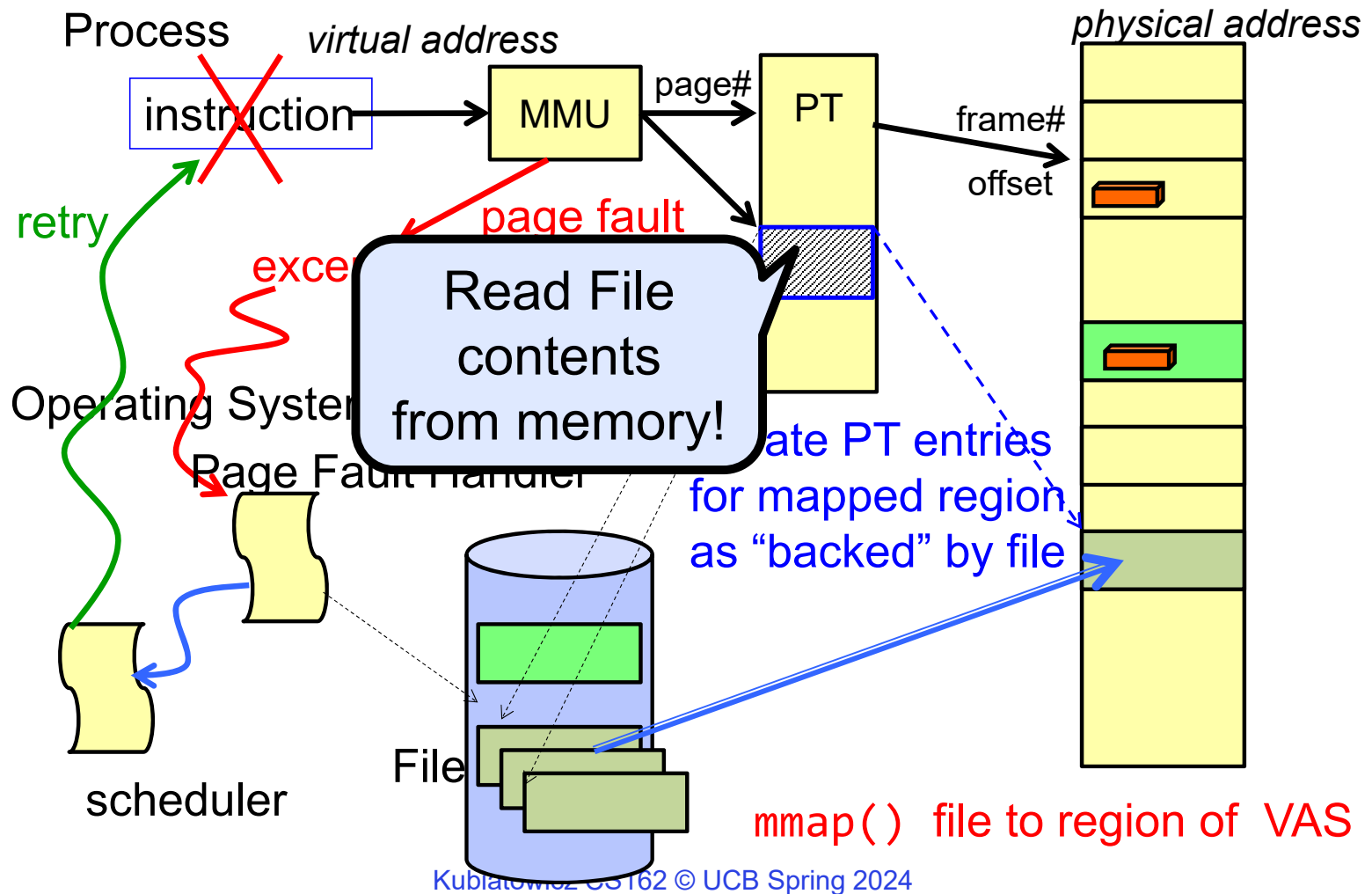
Memory Mapped Files

- Traditional I/O involves explicit transfers between buffers in process address space to/from regions of a file
 - This involves multiple copies into 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); }

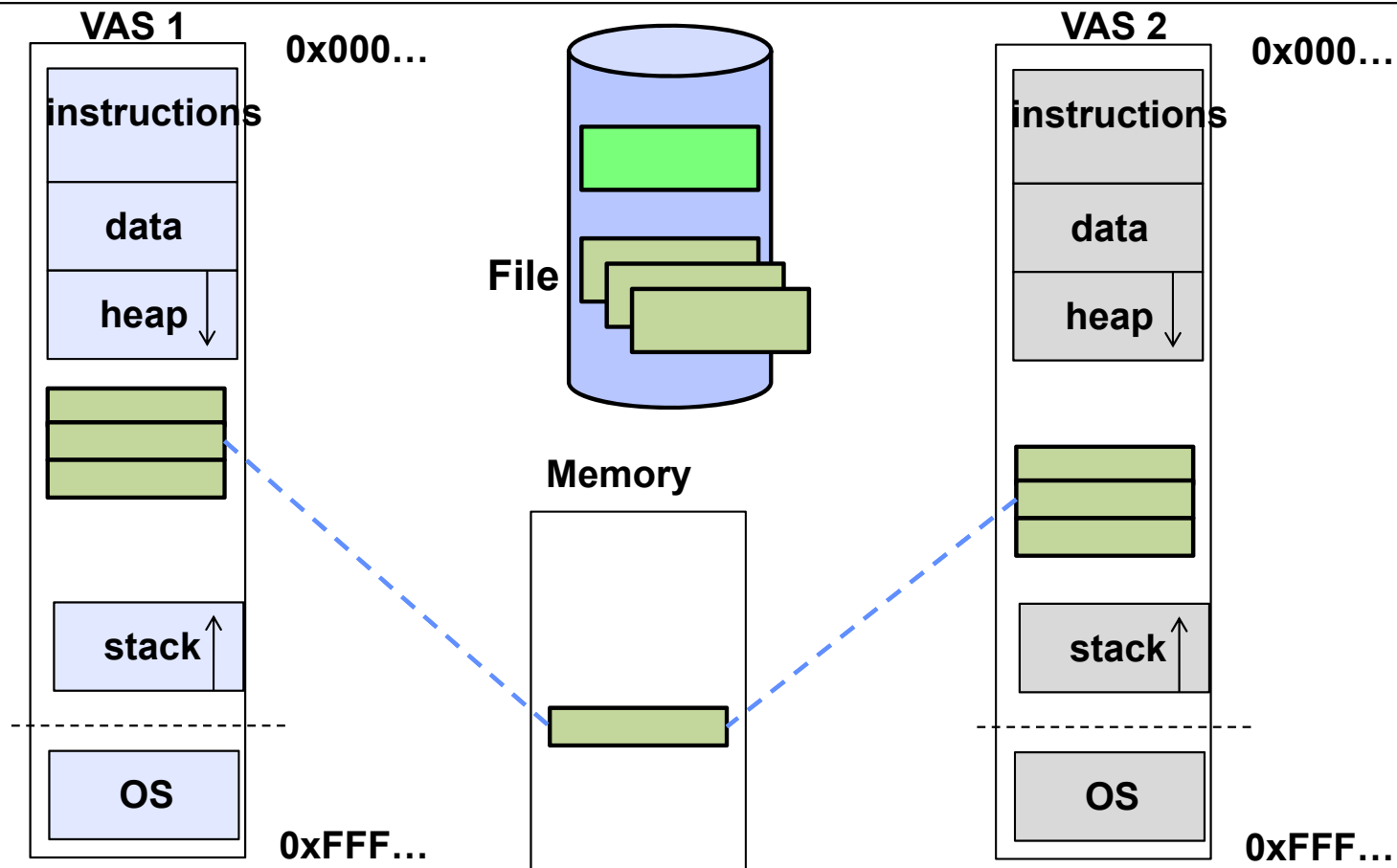
    printf("mmap at : %16lx\n", (long) something);

    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

HOW TO MAKE FILE SYSTEMS MORE *DURABLE?*

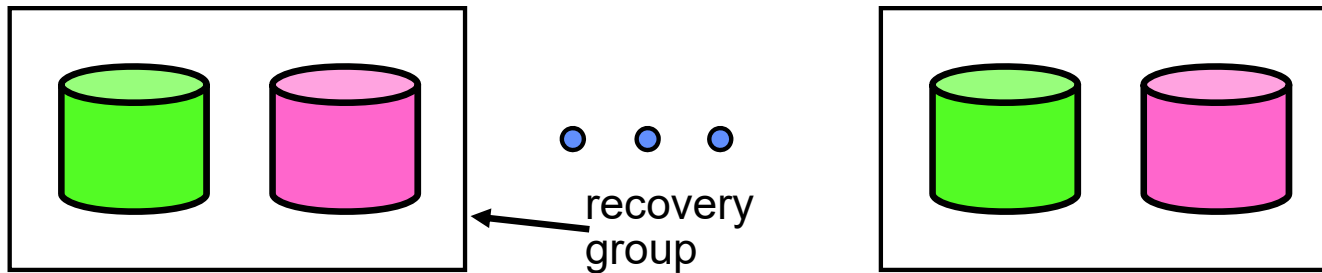
Important “ilities”

- **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

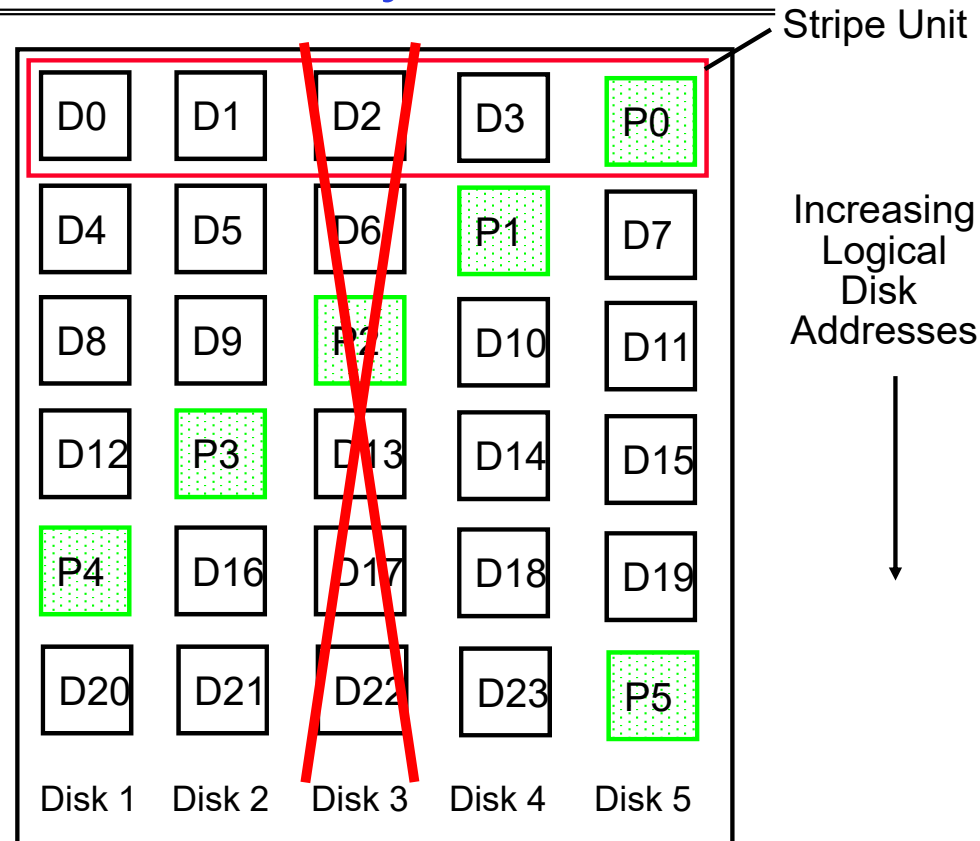


Redundant Array of Inexpensive D
(developed here at Berkeley!)

- 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:
 $D_2 = D_0 \oplus D_1 \oplus D_3 \oplus P_0$

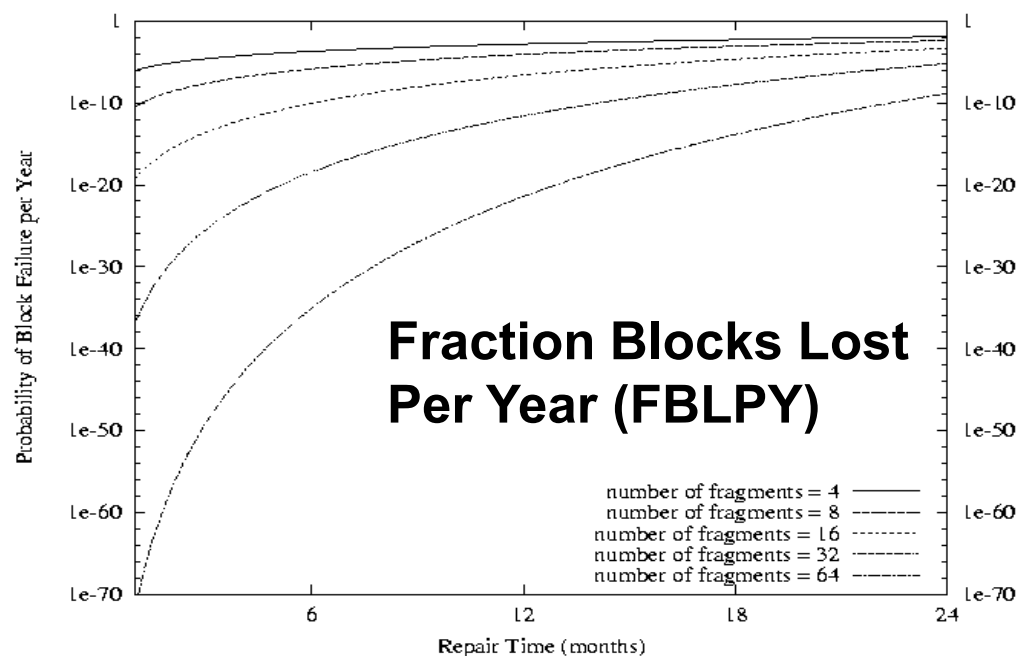


- Can spread information widely across internet for durability
 - RAID algorithms work over geographic scale

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!

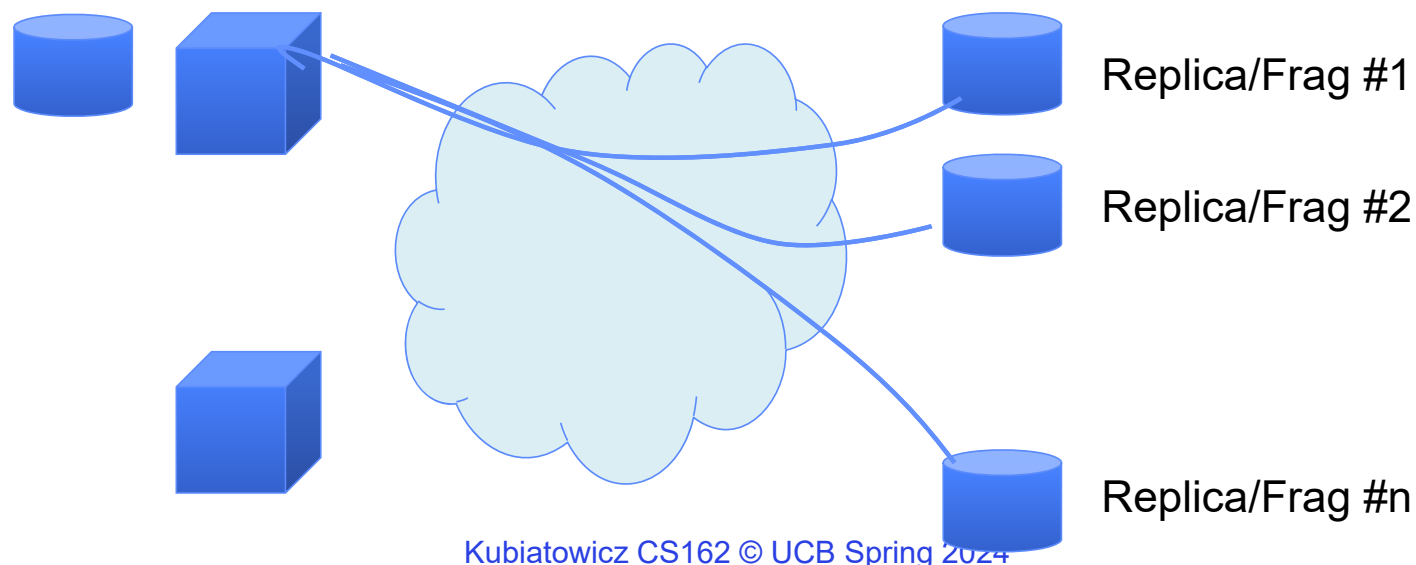
Use of Erasure Coding for High Durability/overhead ratio!



- Exploit law of large numbers for durability!
- 6 month repair, FBLPY with 4x increase in total size of data:
 - Replication (4 copies): 0.03
 - Fragmentation (16 of 64 fragments needed): 10^{-35}

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



File System Summary (1/2)

- File System:
 - Transforms blocks into Files and Directories
 - Optimize for size, access and usage patterns
 - Maximize sequential access, allow efficient random access
 - Projects the OS protection and security regime (UGO vs ACL)
- File defined by header, called “inode”
- Naming: translating from user-visible names to actual sys resources
 - Directories used for naming for local file systems
 - Linked or tree structure stored in files
- 4.2 BSD Multilevel Indexed Scheme
 - inode contains file info, direct pointers to blocks, indirect blocks, doubly indirect, etc..
 - NTFS: variable extents not fixed blocks, tiny files data is in header

File System Summary (2/2)

- File layout driven by freespace management
 - Optimizations for sequential access: start new files in open ranges of free blocks, rotational optimization
 - Integrate freespace, inode table, file blocks and dirs into block group
- 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)