CS162
Operating Systems and Systems Programming
Lecture 20

File Systems & Reliability

Professor Natacha Crooks & Matei Zaharia

https://cs162.org/

Slides based on prior slide decks from David Culler, Ion Stoica, John Kubiatowicz, Alison Norman and Lorenzo Alvisi

Recall: Unix File System

Superblock object: information about file system

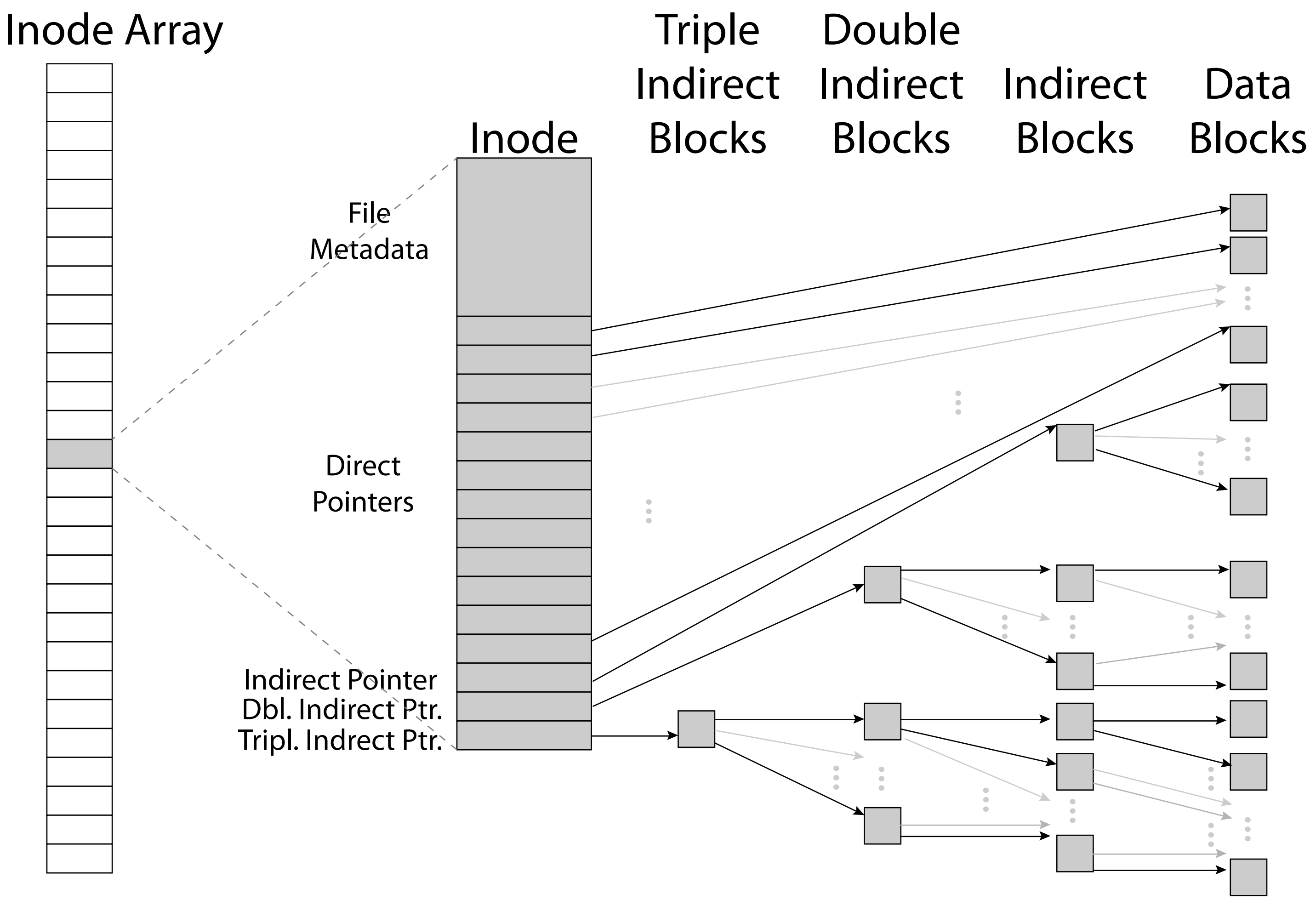
Free bitmaps: what is allocated/not allocated

Inode object: represents a specific file on disk

Dentry object: directory entry, single component of a path

Blocks: How files are stored on disk

Recall: Inode Data Structure



Recall: Fast File System

Same inode structure as in BSD 4.1

- Same file header and triply indirect blocks like we just studied
- Some changes to block sizes from 1024⇒4096 bytes for performance

Optimization for Performance and Reliability:

- Distribute inodes among different tracks to be closer to data
- Uses bitmap allocation in place of freelist
- Attempt to place files contiguously (look for big empty ranges when appending)
- 10% reserved disk space
- Skip-sector positioning

Plan for Today

Quick examples of two other file systems

Buffer cache (memory cache for FS)

Reliability

Other File Systems

FAT:
File Allocation Table

(MS-DOS,1977)

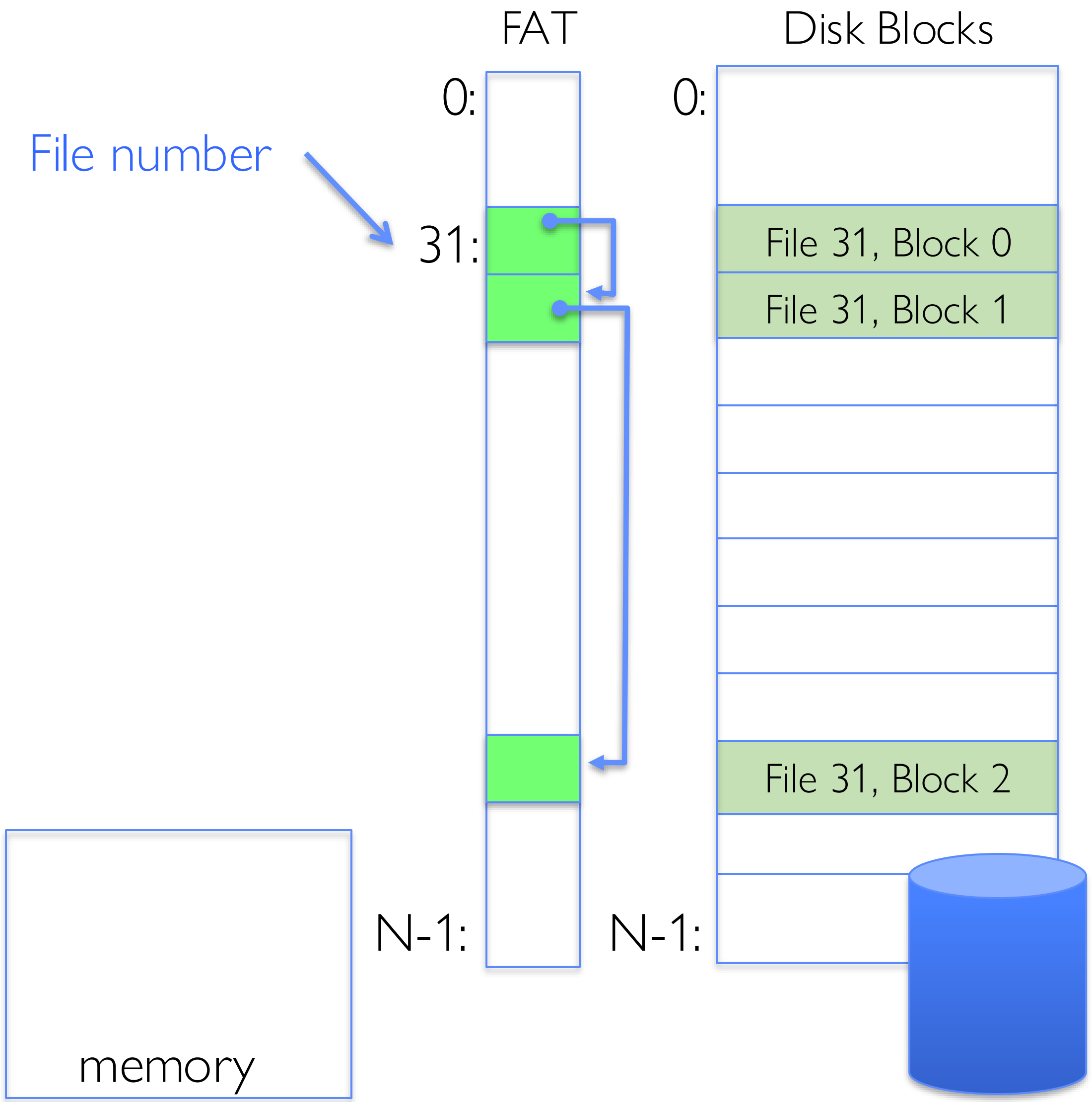
Windows NTFS

FAT (File Allocation Table) File System

Example:

file_read 31, < 2, x >

- Index into FAT with file number
- Follow linked list to block
- Read the block from disk into memory



FAT (File Allocation Table)

File is a collection of disk blocks

FAT is linked list 1-1 with blocks

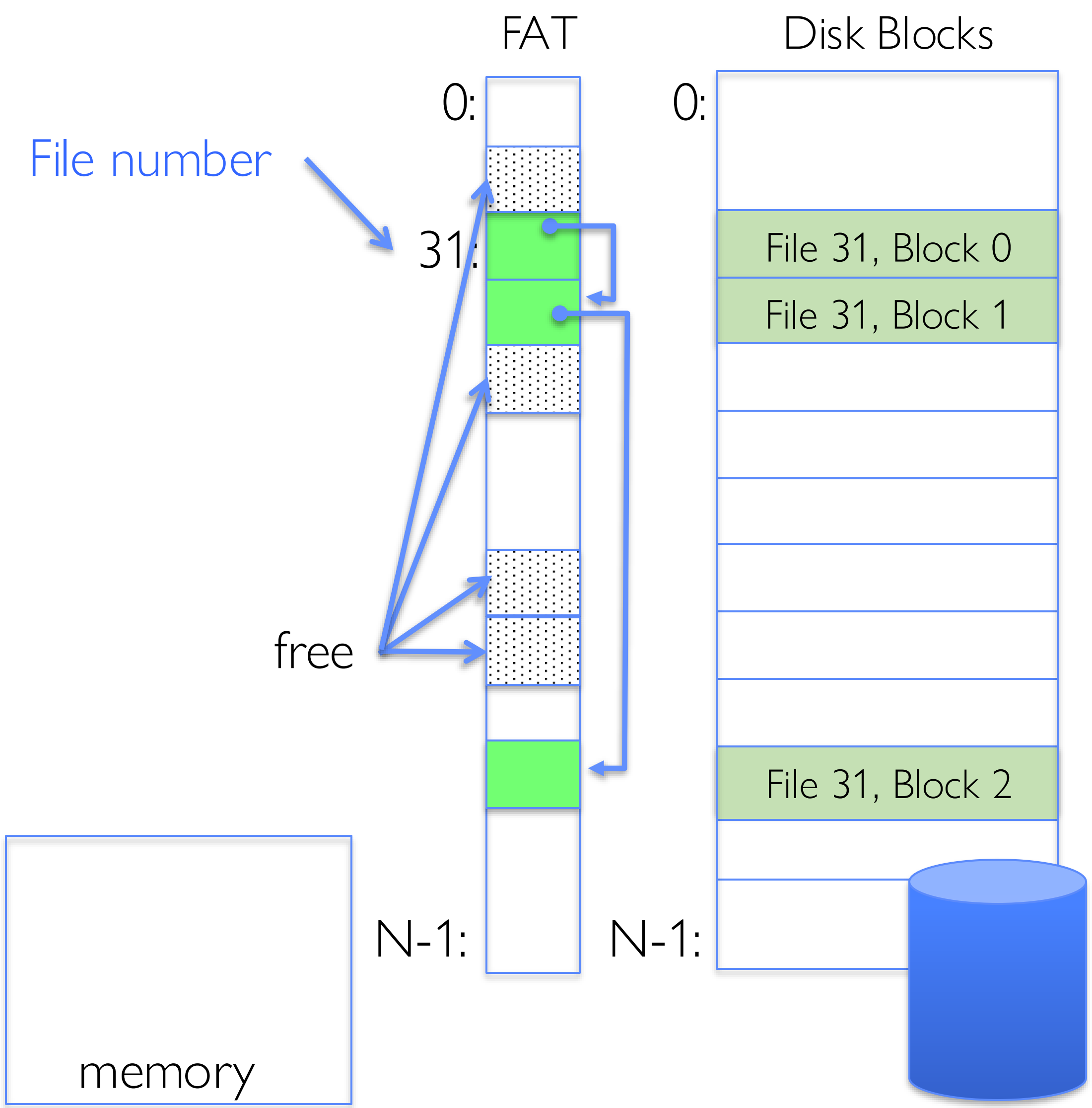
File number is index of root of its block list

File offset: block number + offset in block

Follow list to get block number

Unused blocks marked free

- Could require scan to find
- Or, could use a free list

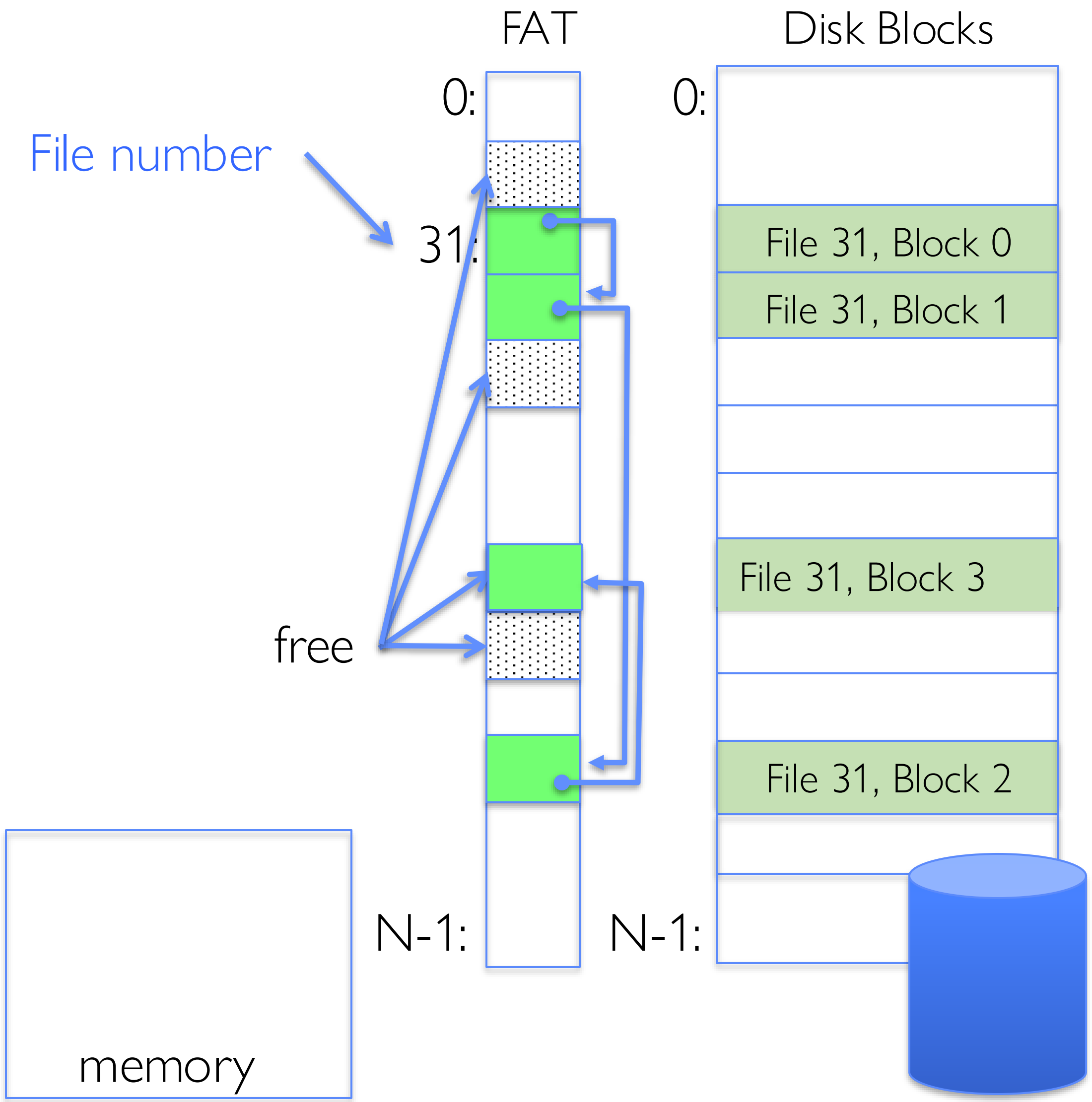


FAT (File Allocation Table)

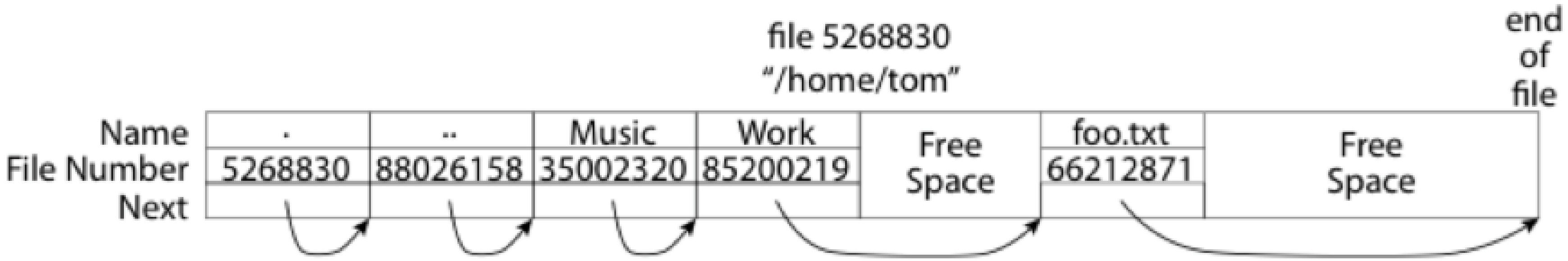
file_write(31, < 3, y >)

– Grab free block

– Linking them into file



Directories in FAT



A directory is a file containing <file_name: file_number> mappings

In FAT: file attributes are kept in directory (!!!)

– Not directly associated with the file itself

Each directory has a linked list of entries

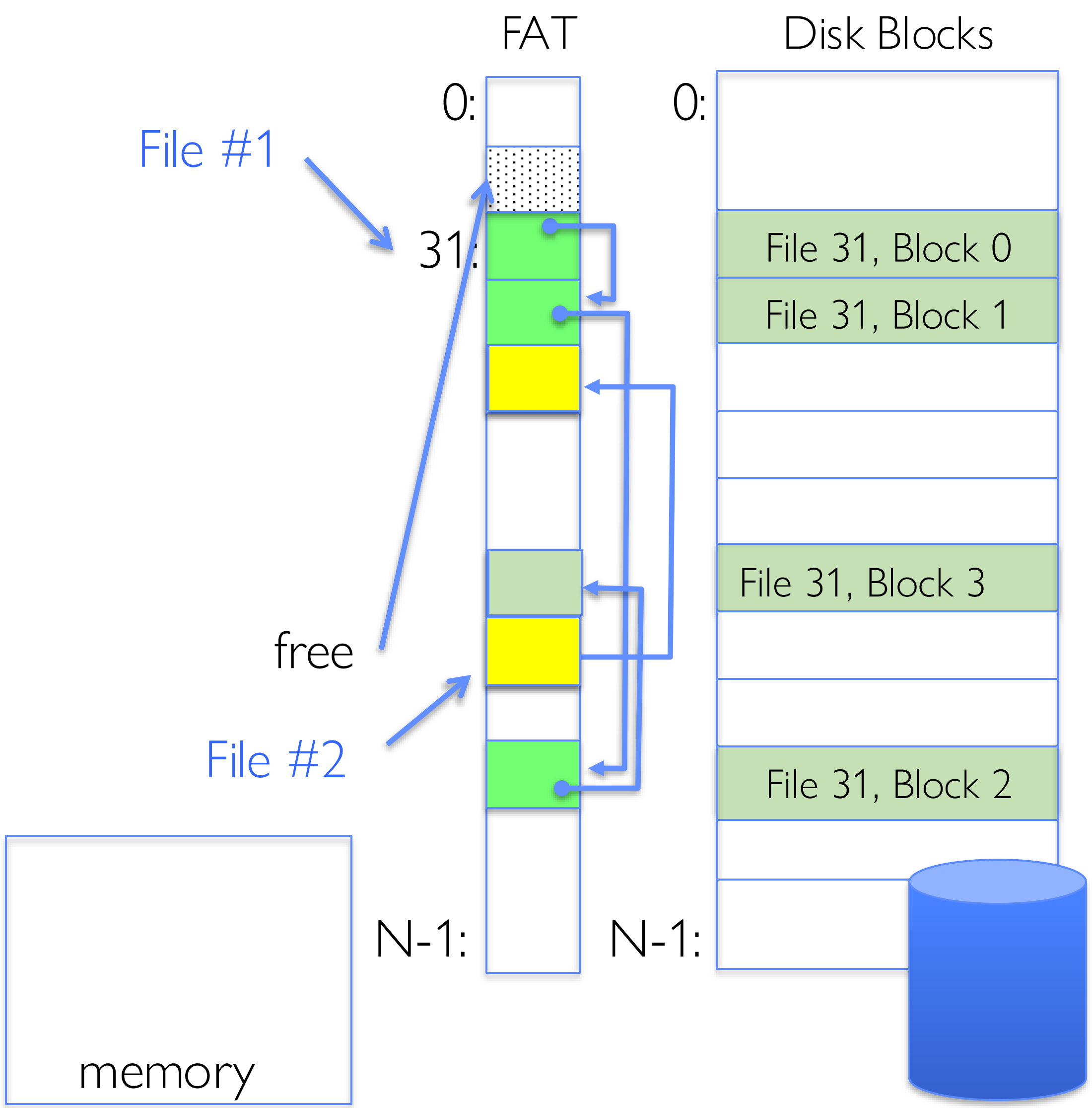
– Requires linear search to find an entry

Root directory is always at block 2

FAT Discussion

Suppose you start with the file number:

- Time to find first block?
- Sequential access?
- Random access?
- Fragmentation?



Windows New Technology File System (NTFS)

Default on recent Windows systems

Variable length *extents* (typically one or more blocks)

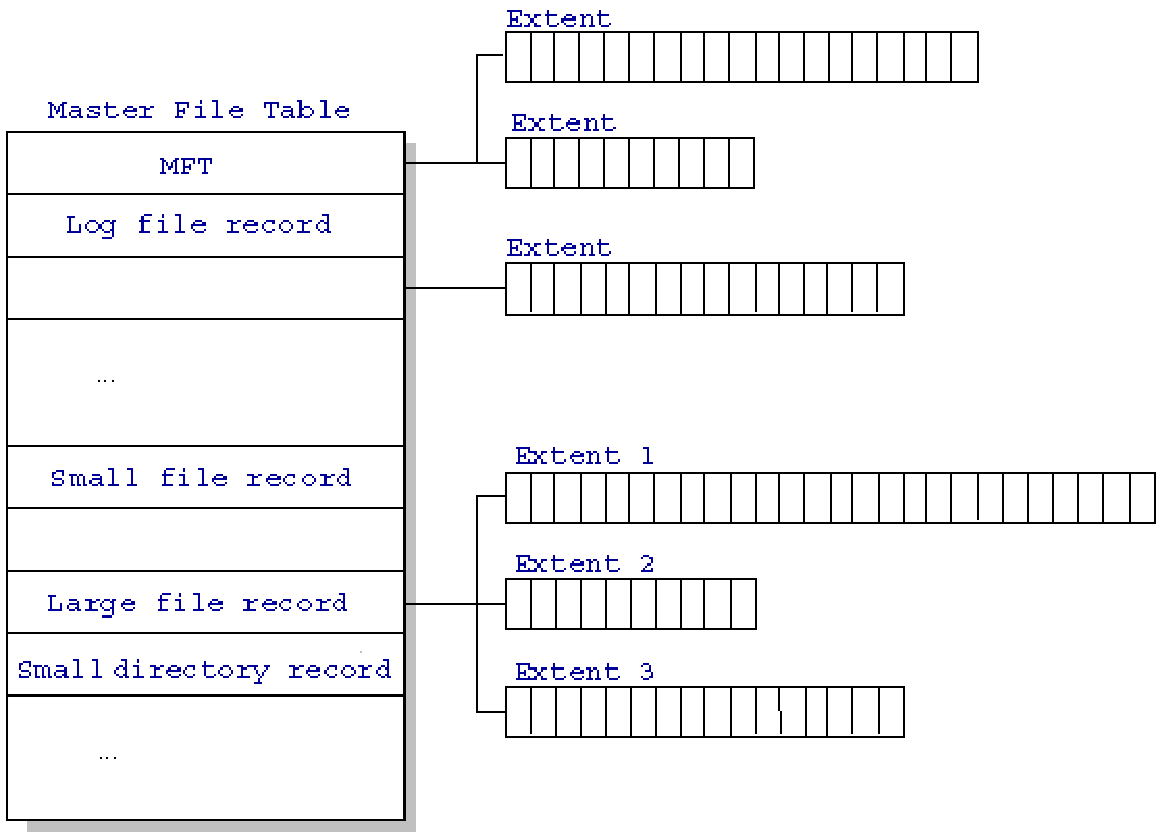
Master File Table

- Metadata area with 1KB long entries
- Every entry is (mostly) a sequence of <attribute:value>

Each entry in MFT contains metadata and:

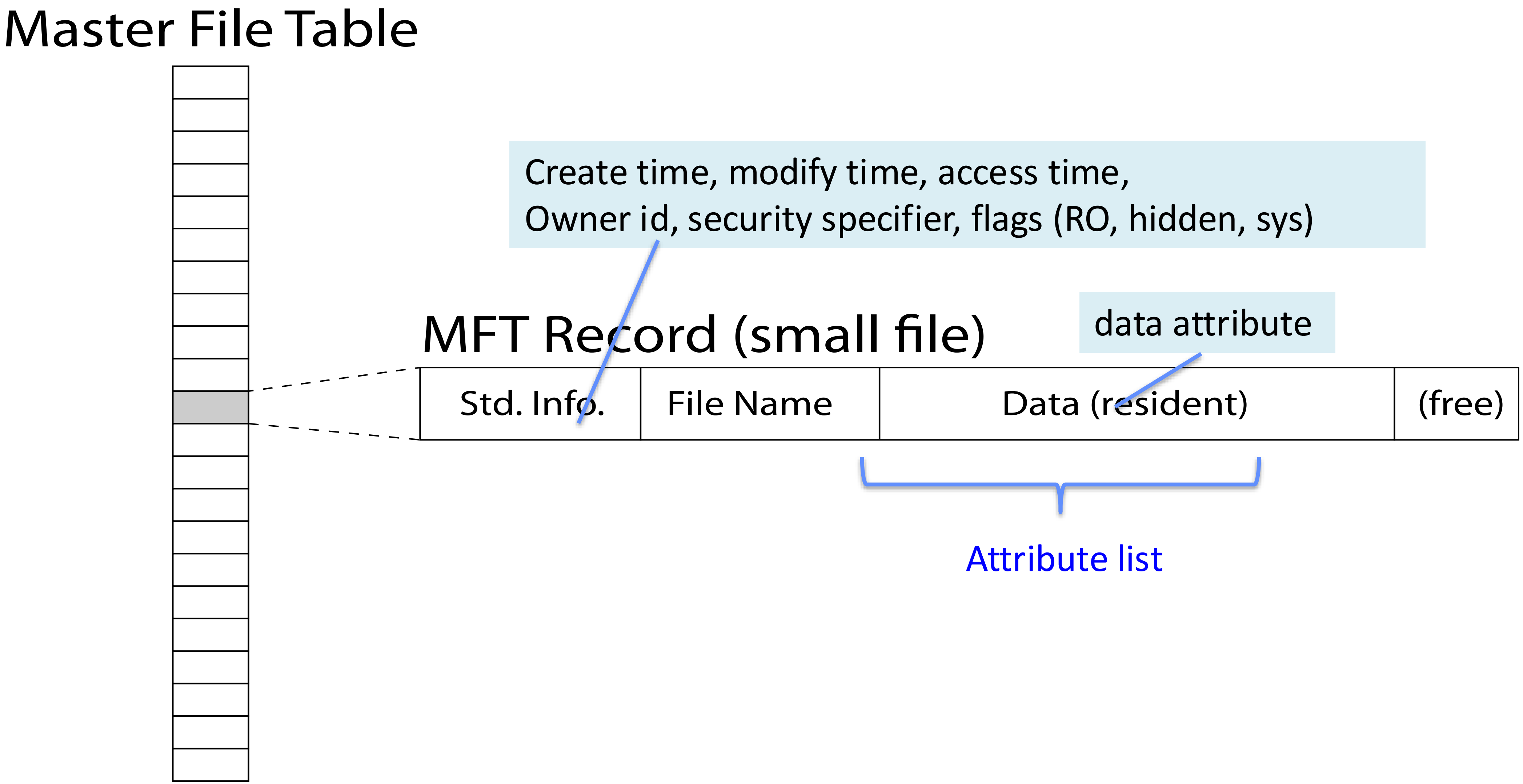
- File's data directly (for small files)
- A list of extents (start block, size) for file's data
- For very big files: pointers to other MFT entries with *more* extent lists

NTFS

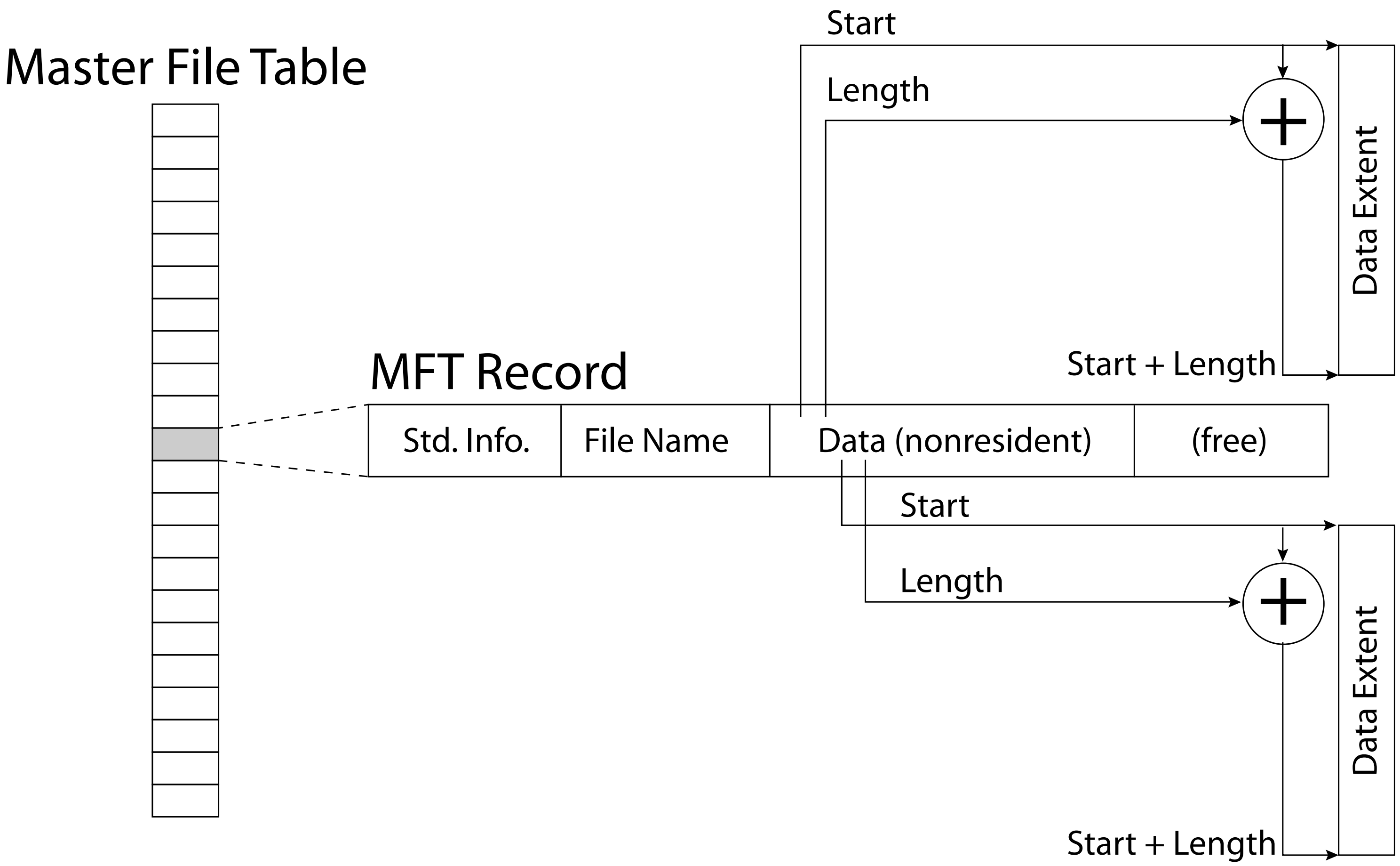


http://ntfs.com/ntfs-mft.htm

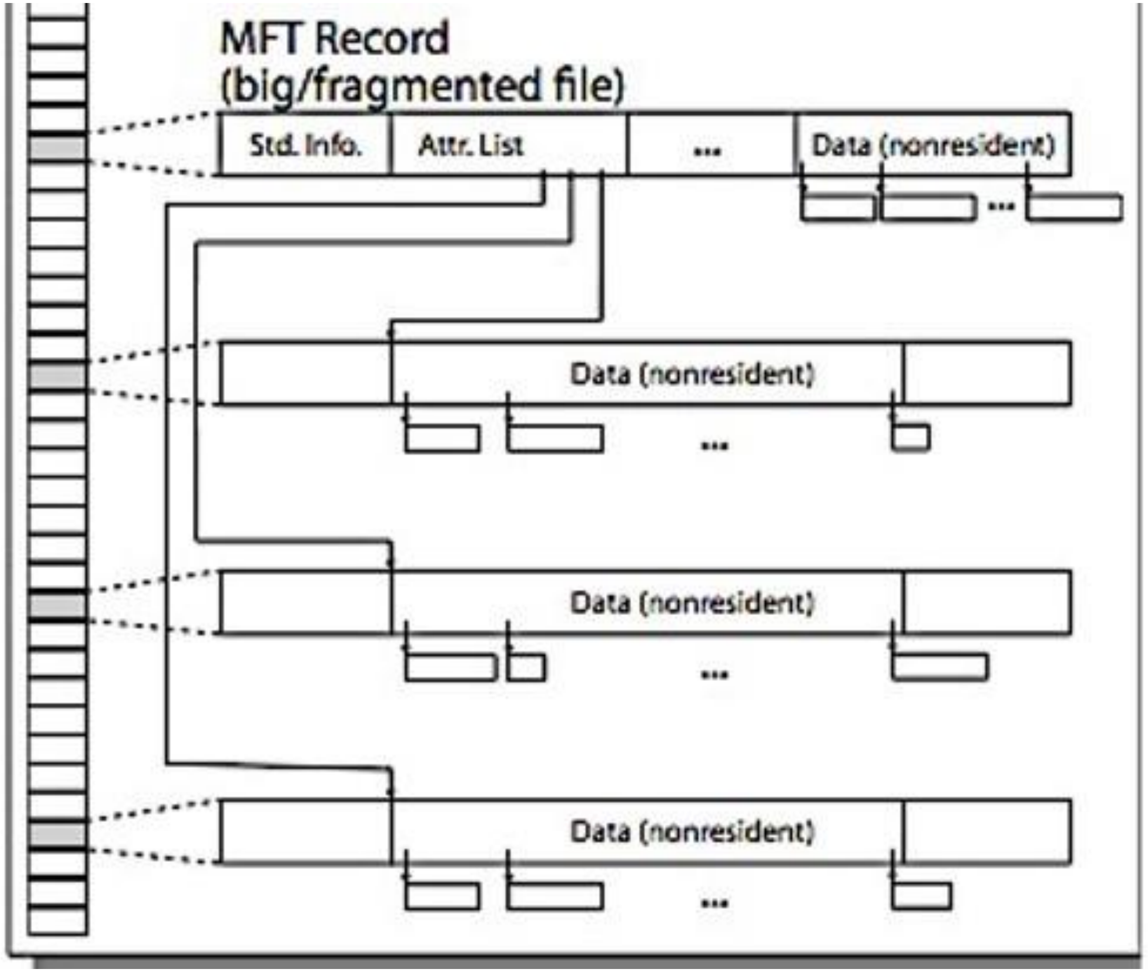
NTFS Small File: Data stored with Metadata



NTFS Medium File: Extents for File Data



NTFS Large File: Pointers to Other MFT Entries



Directories in NTFS

Implemented as B-Trees

File's number identifies its entry in MFT

MFT entry for each file also has a file name attribute

– Hard links require multiple file name attributes in the MFT entry

Buffer Caches

Kernel *must* copy disk blocks to main memory to access their contents and write them back if modified

Key Idea: Exploit temporal locality by caching disk data in memory

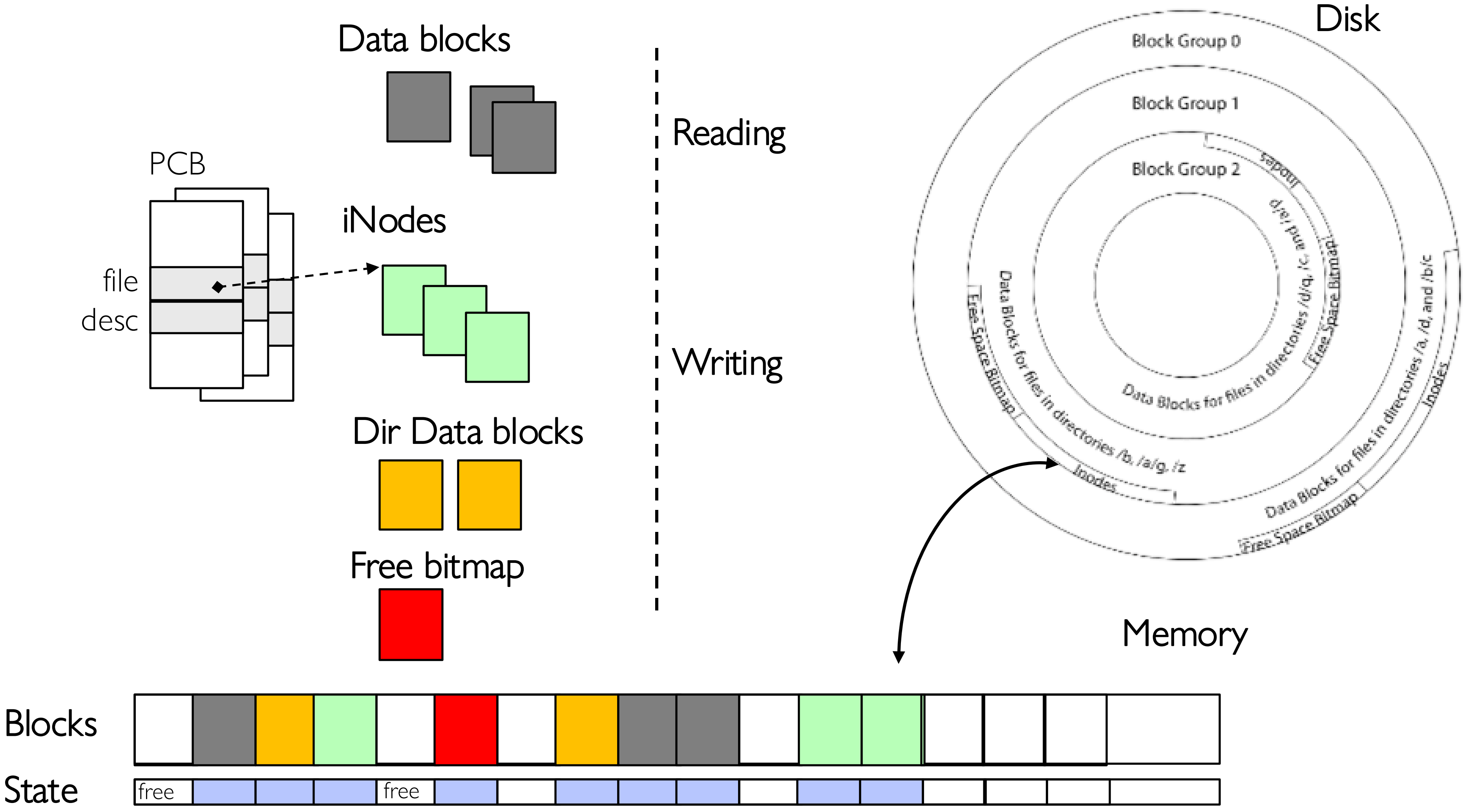
- Disk blocks: Mapping from block address$\rightarrow$disk content
- Name translations: Mapping from paths$\rightarrow$inodes

Buffer Cache: Memory used to cache FS data, including disk blocks and metadata

- Can contain “dirty” blocks (with modifications not on disk)

File System Buffer Cache

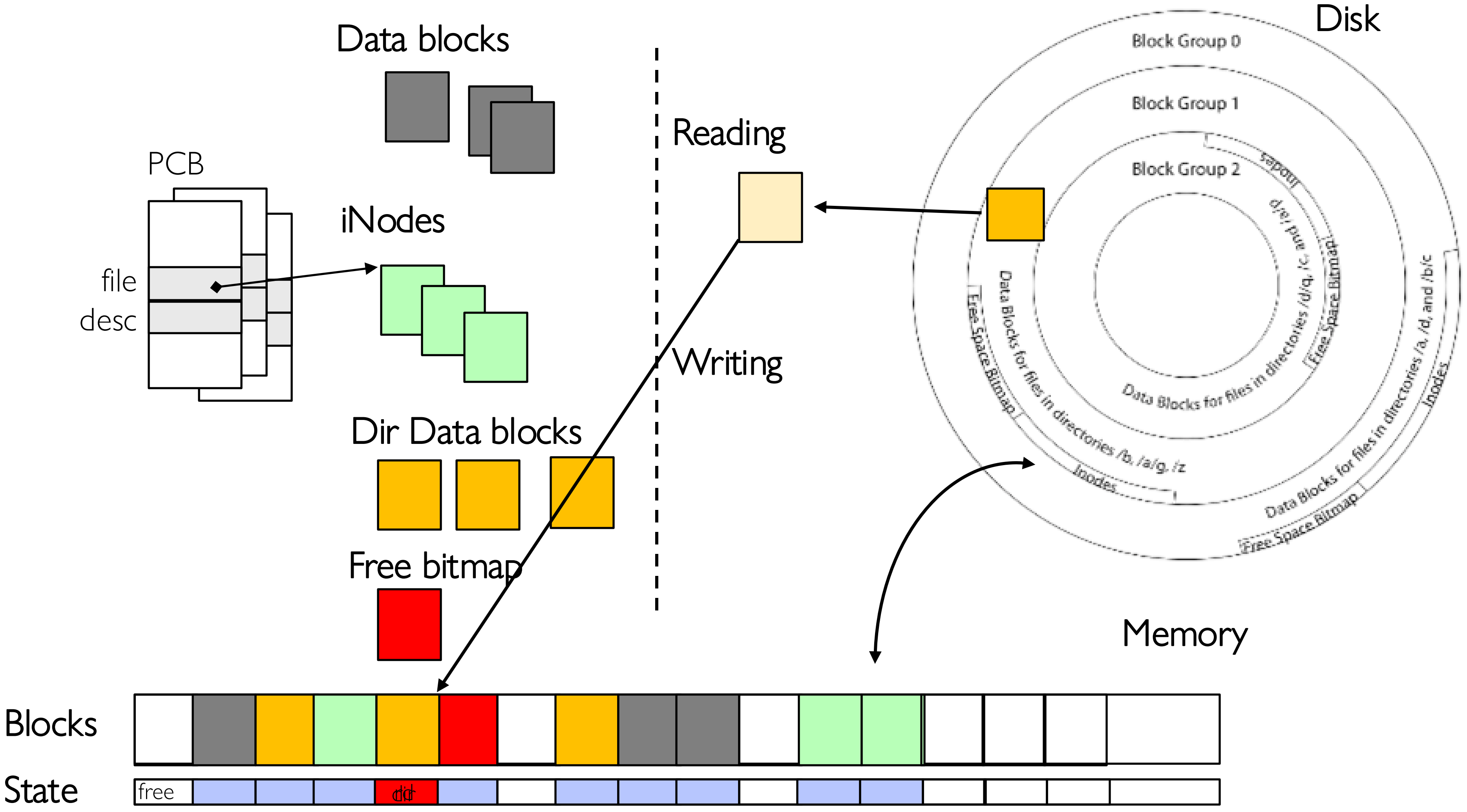
OS implements a cache of disk blocks for efficient access to data, directories, inodes, freemap



File System Buffer Cache: open

Directory lookup (repeat as needed):

- load block of directory
- search for map

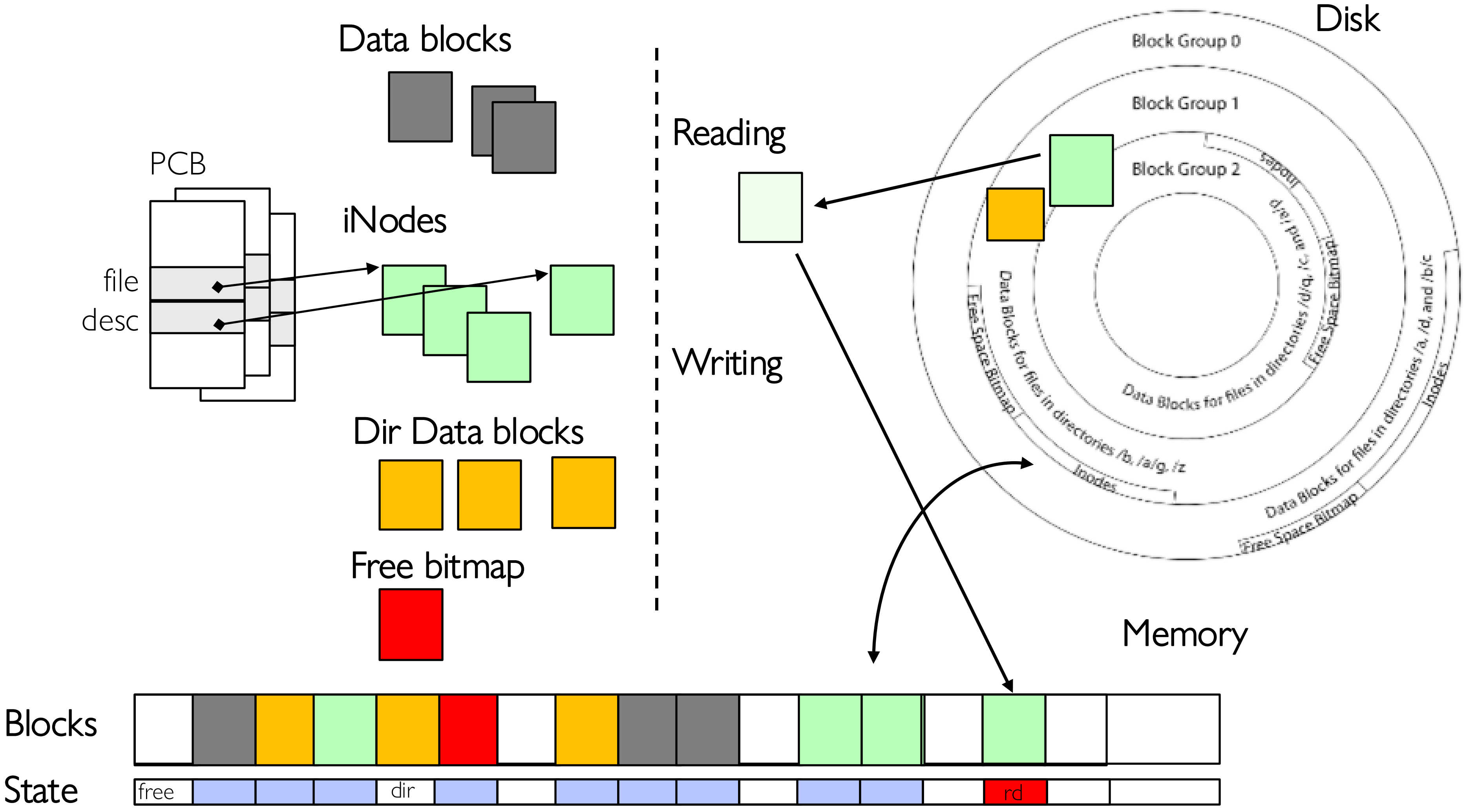


File System Buffer Cache: open

Directory lookup (repeat as needed):

- load block of directory
- search for map

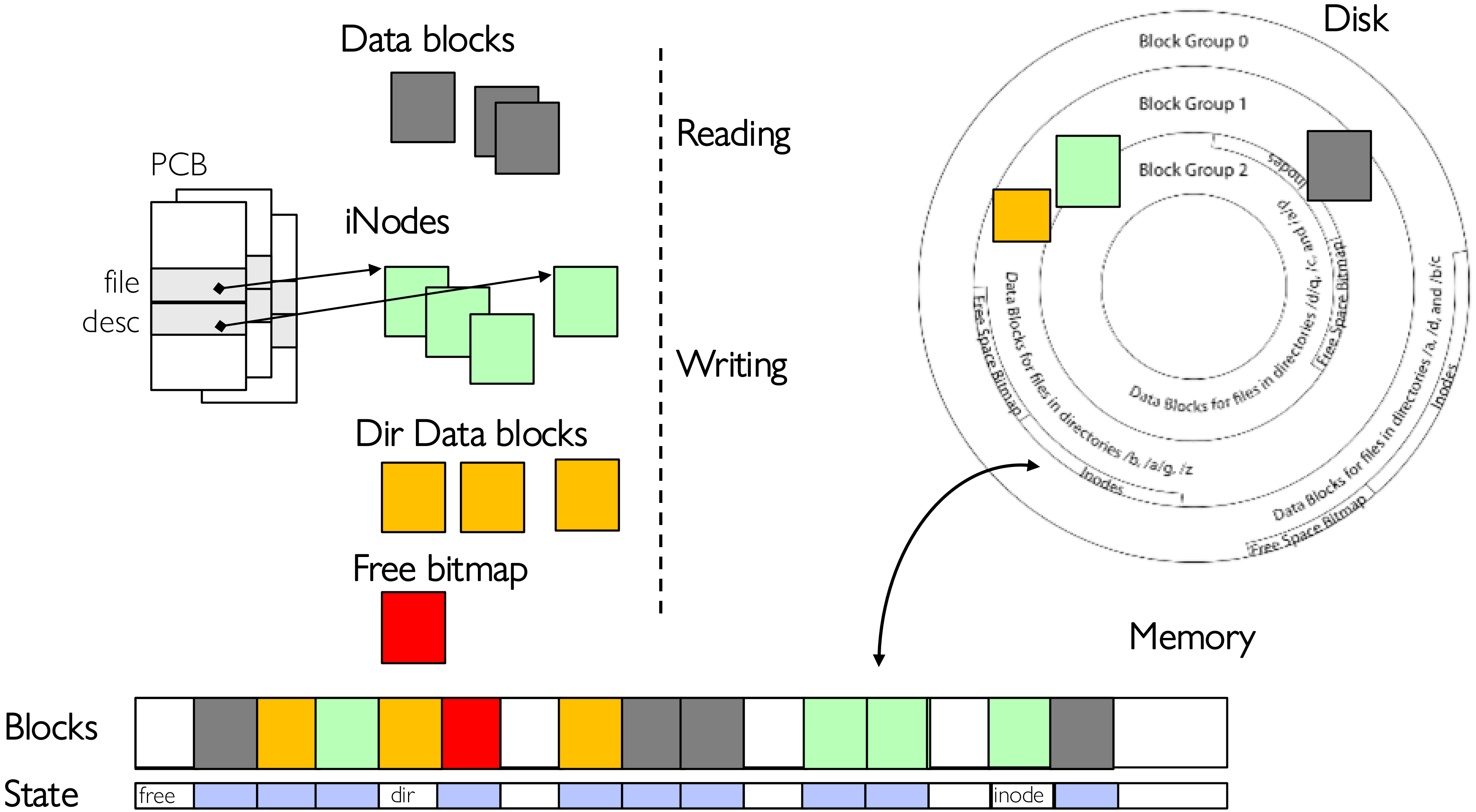
Create reference in open file descriptor



File System Buffer Cache: read?

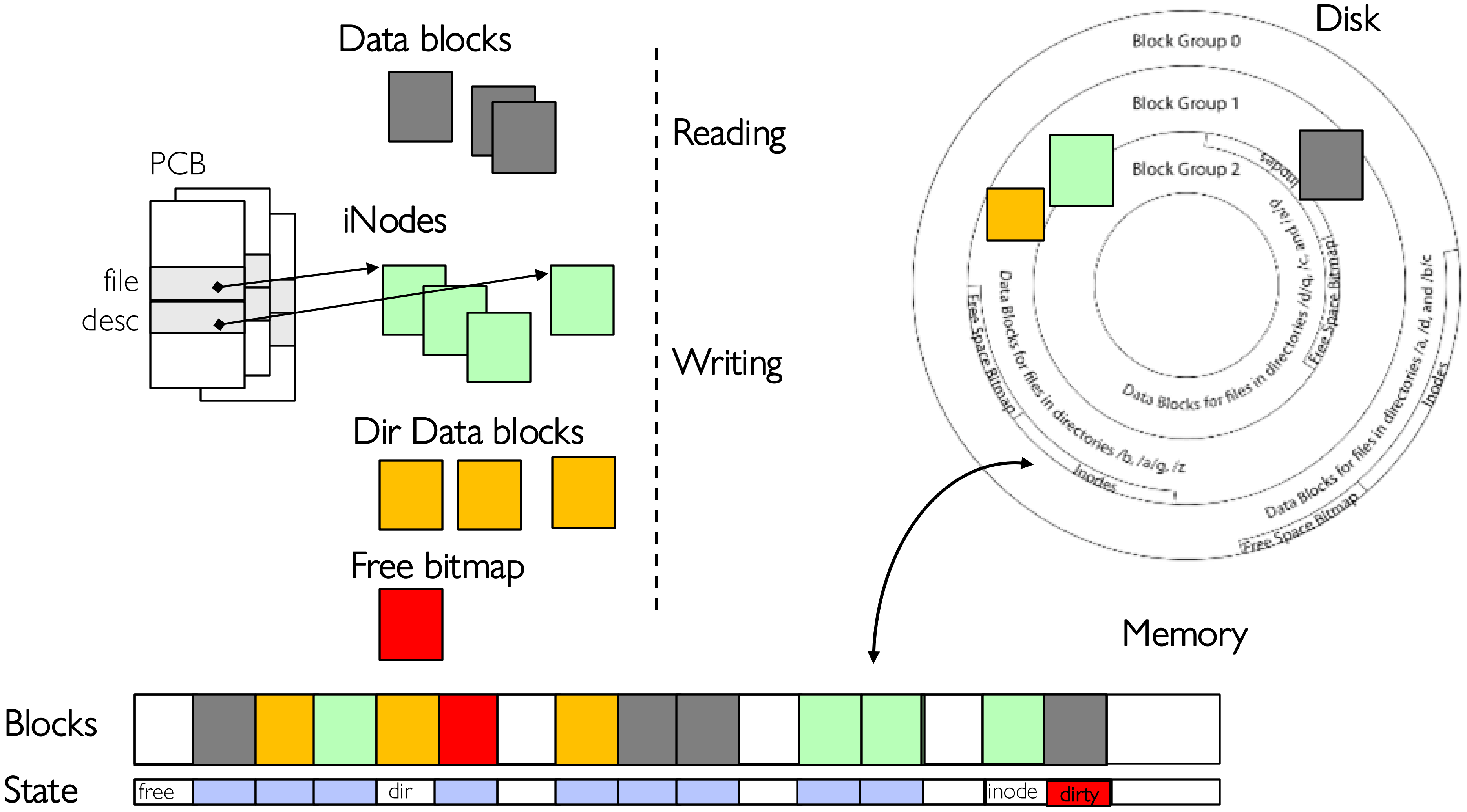
Read Process

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



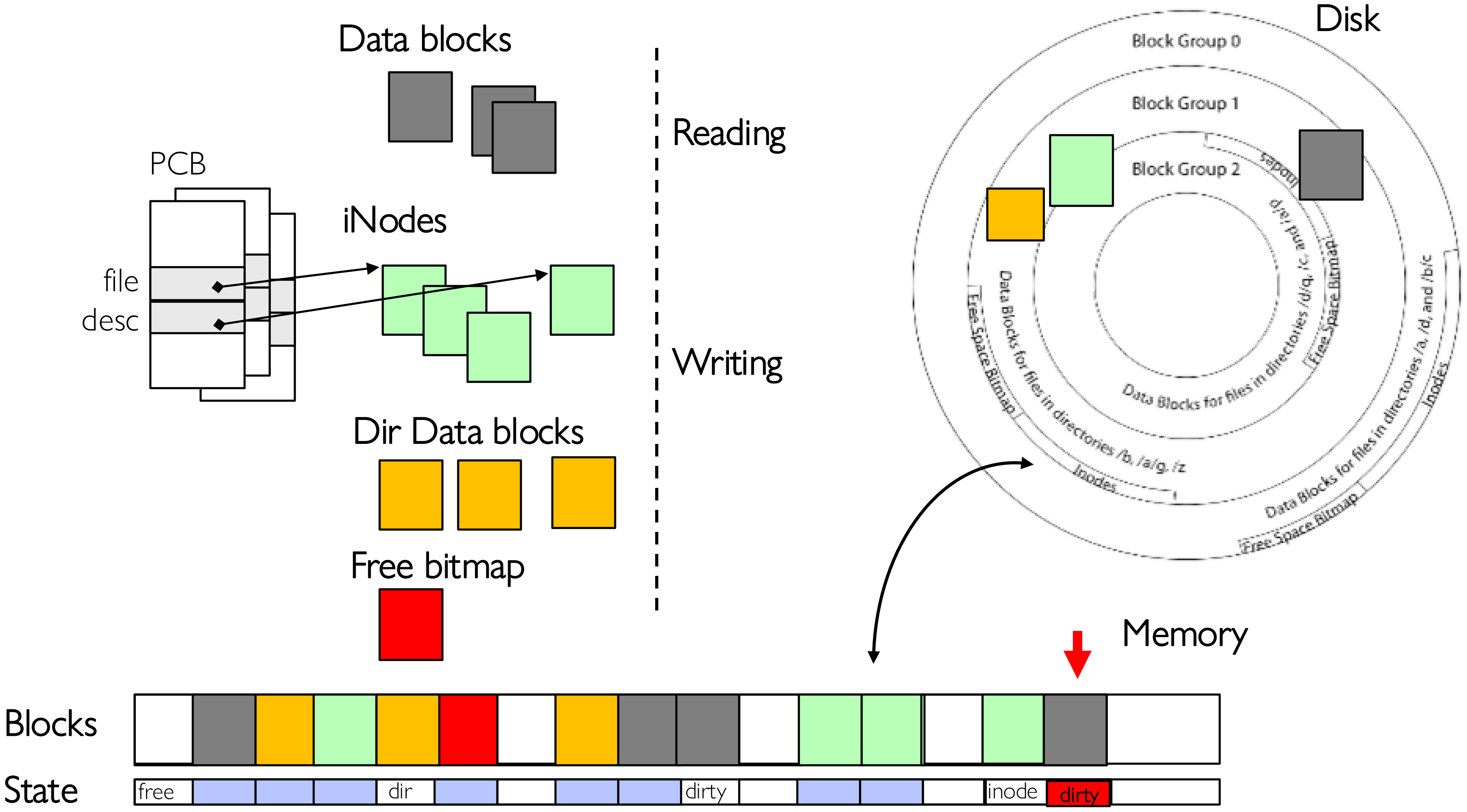
File System Buffer Cache: write?

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



File System Buffer Cache: Eviction?

Blocks being written back to disk go through a transient state



Buffer Cache Replacement Policy

Preferred 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

Buffer 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

`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)

Advantage of Delayed Writes

Latency: 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 writes 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
- Applications can use `fsync` to force flushing a file

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

Boom!


HANDLED*** Address 8016a950 has base at 80100000
.6.2 irql:1f SYSVER 0xf0000565
Name
Dll Base DateStmp - Name
ntoskrnl.exe
atapi.sys
Disk.sys
Ntfs.sys
NTice.sys
Cdrom.SYS
KSecDD.SYS
win32k.sys
Cdfs.SYS
nbf.sys
netbt.sys
afd.sys
Parport.SYS
80010000
80007000
801db000
80237000
f1f48000
f228c000
f2290000
fe0c2000
fdca2000
fdc35000
f1f68000
f2008000

Important "ilities"

Availability

The probability that the system can accept and process requests

Durability

The ability of a system to recover data despite faults

Reliability

The ability of a system or component to perform its required functions under stated conditions for a specified period of time (IEEE definition)

File System 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

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 $\Leftarrow$ inode $\Leftarrow$ free $\Leftarrow$ 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

Reliability Approach #1: Careful Ordering

Sequence operations in a specific order
- Careful design to allow sequence to be interrupted safely

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

Create a new version of the file with the updated data
- Reuse blocks that don't change much of what is already in place
- Only point to the new version when fully done writing it

Seems expensive!
- But updates can be batched, and many 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

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 one 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: Transactions in SQL

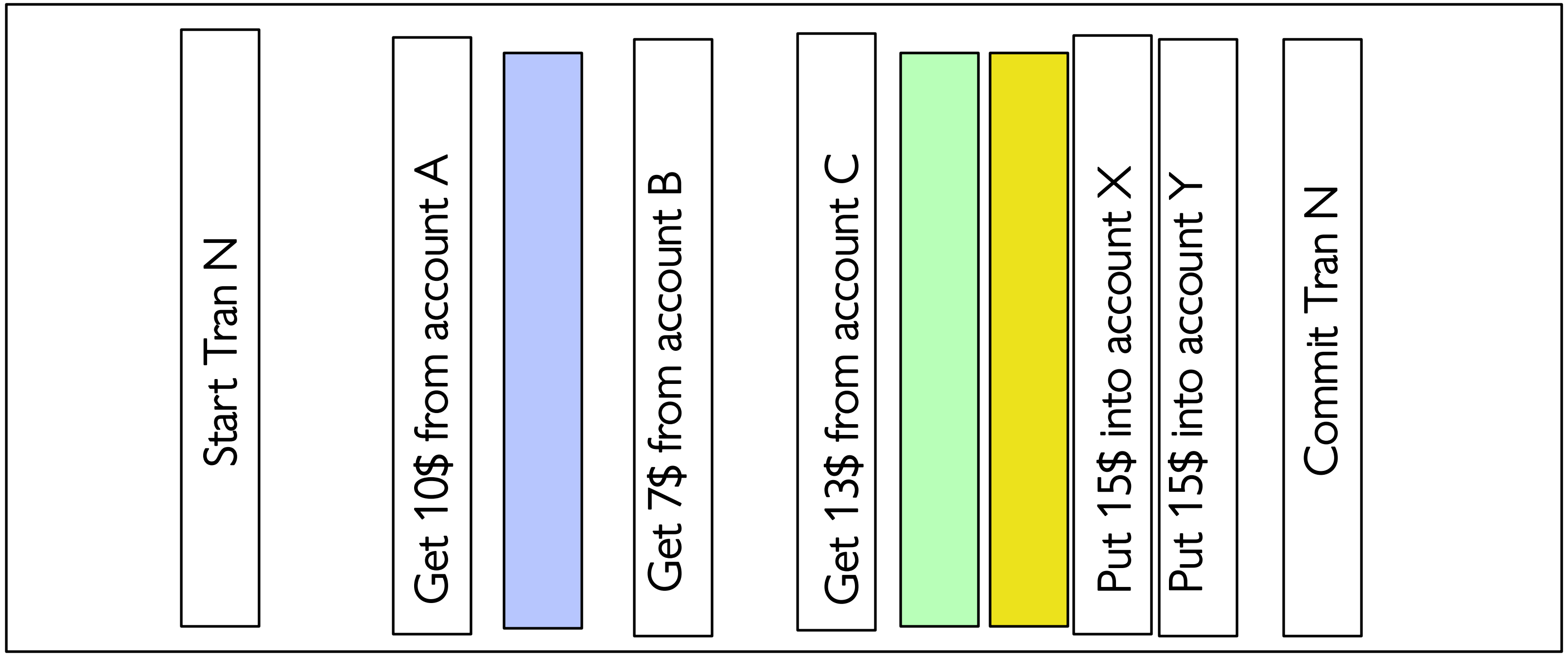
```
BEGIN;    --BEGIN TRANSACTION
 UPDATE accounts SET balance = balance - 100.00 WHERE
   name = 'Alice';

 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

Useful Tool: 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 to all FS data structures are treated as transactions
- A transaction is committed once it is fully written to the log
  - » Data forced to disk for reliability
  - » Process can be accelerated with NVRAM
- Although the actual file system data structures may not be updated immediately, data preserved in the log and replayed to recover

Difference between “Log Structured” and “Journaled” file systems

- In a Log Structured filesystem, data stays in log form
- In a Journaled filesystem, log only 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 specific blocks for it)

Once changes are in the log, they can be safely applied to other data structures

- E.g. modify inode pointers and directory entries

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

Creating a File (No Journaling Yet)

Find free data block(s)

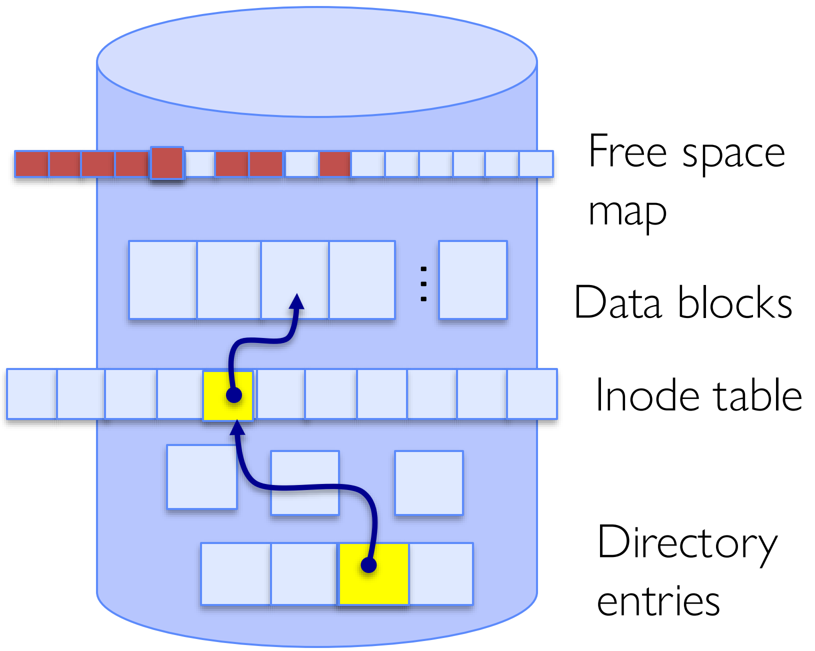
Find free inode entry

Find dirent insertion point

Write bitmap (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)

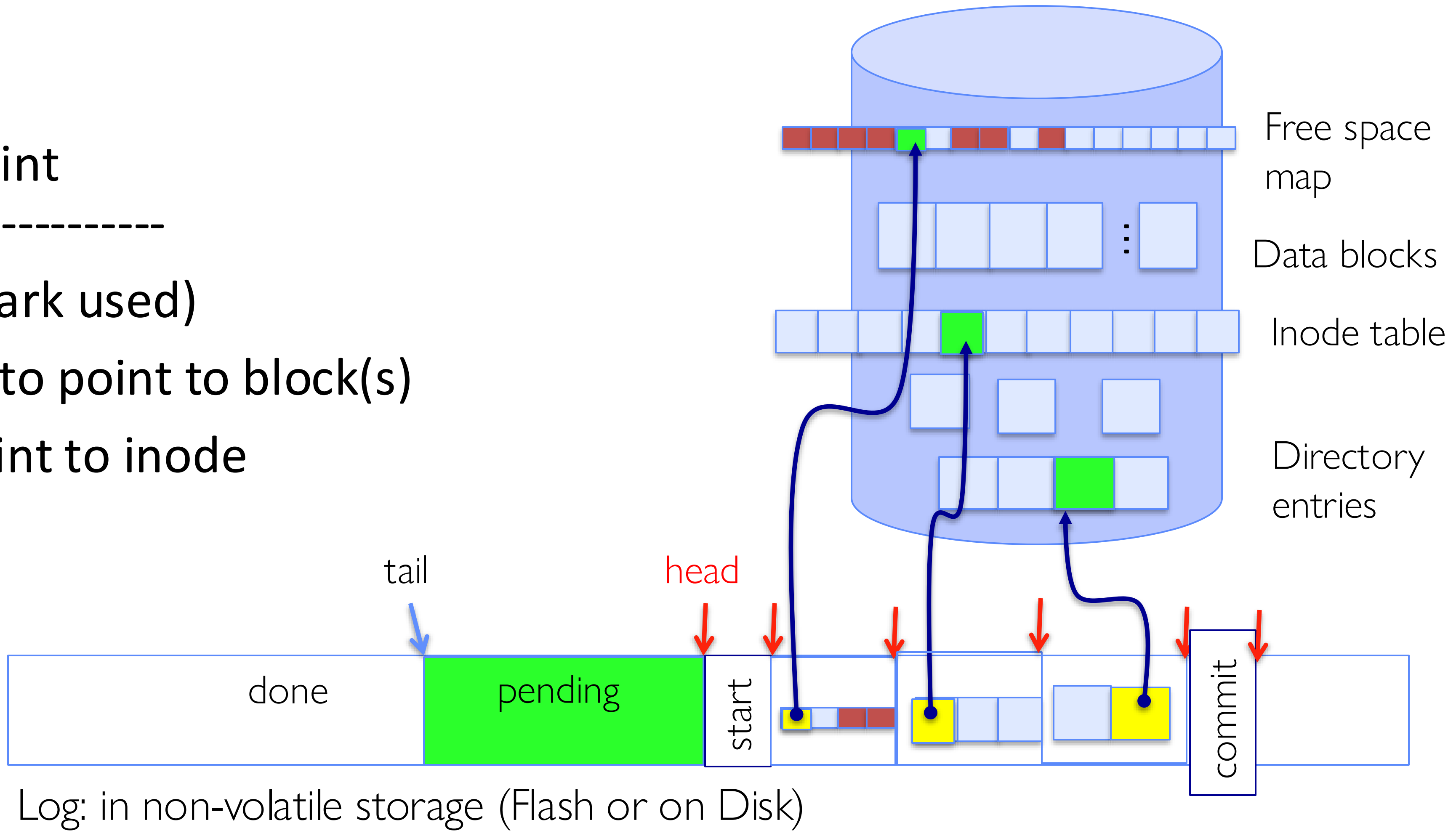
Find free inode entry

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

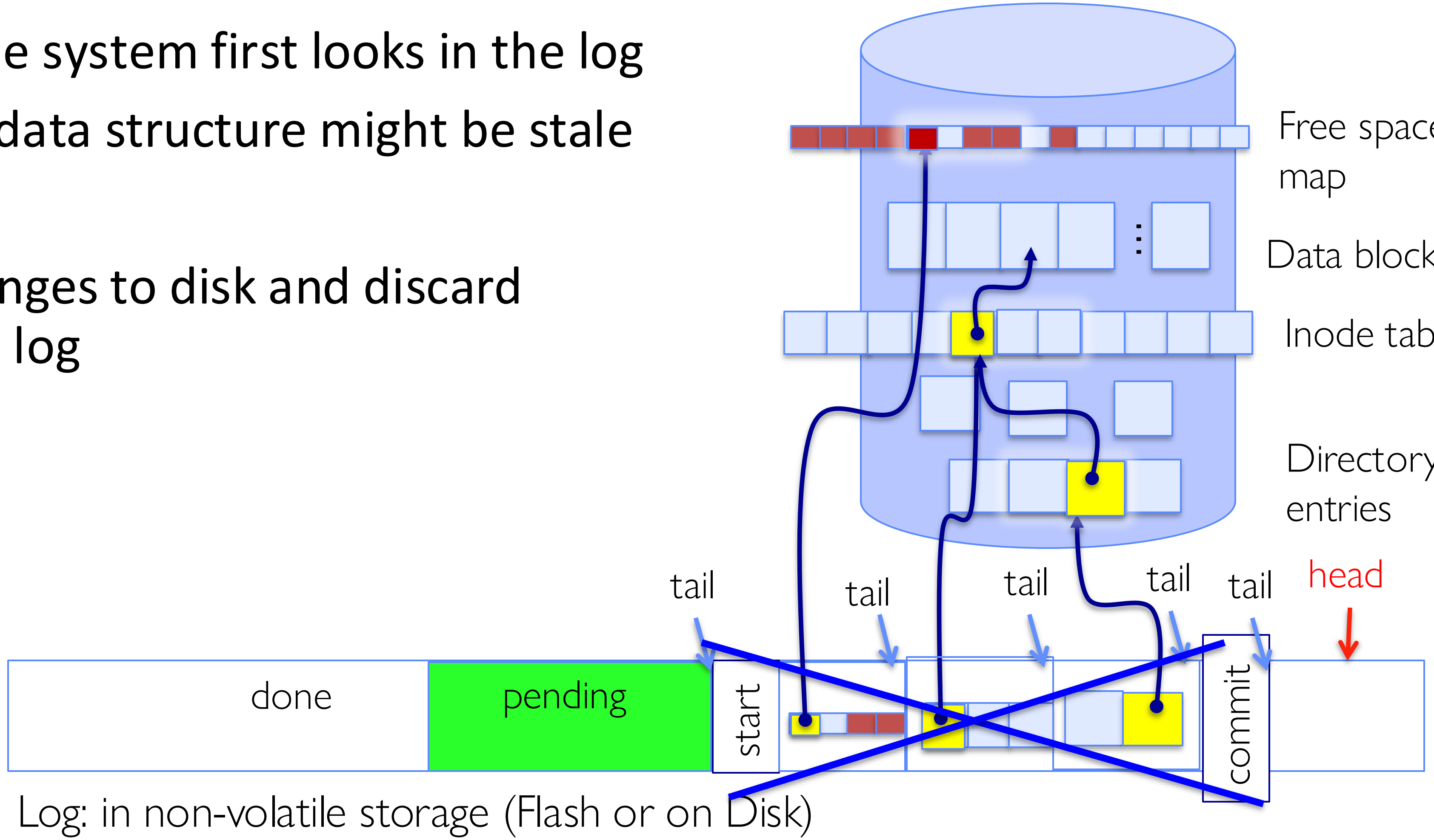


After Commit, Eventually Replay Transaction

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



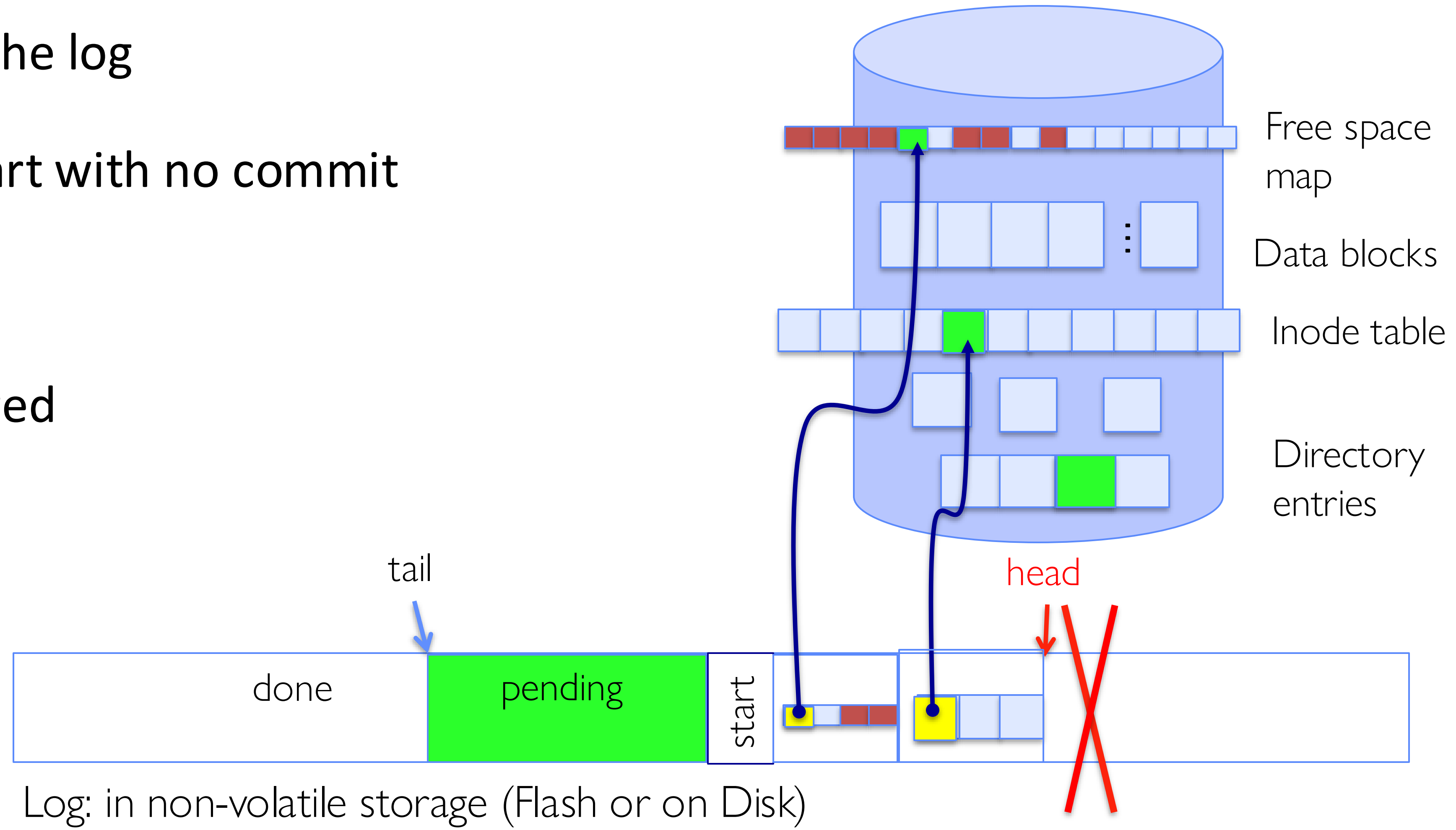
Crash Recovery: Discard Partial Transactions

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)

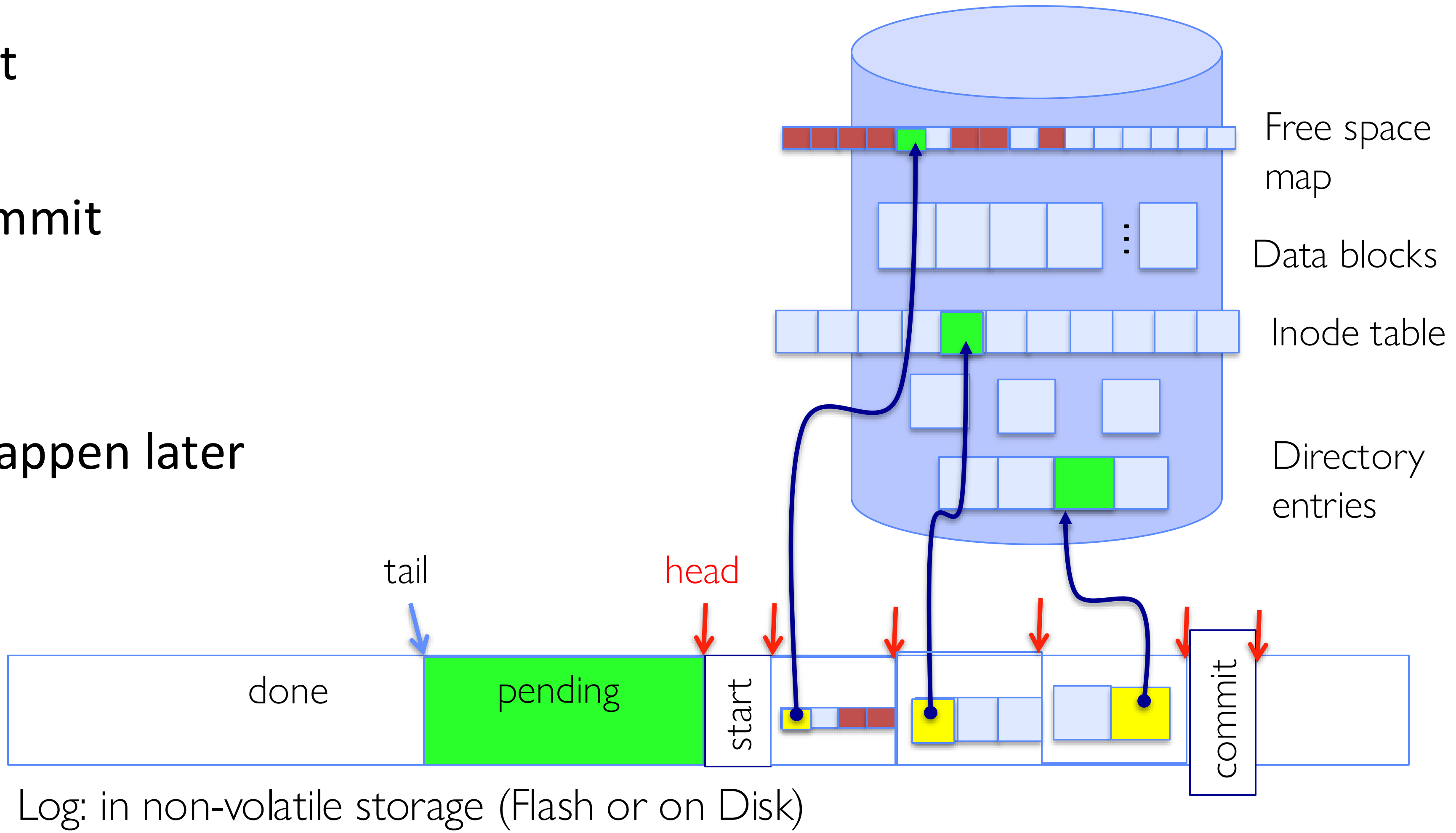
Crash Recovery: Keep Complete Transactions

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