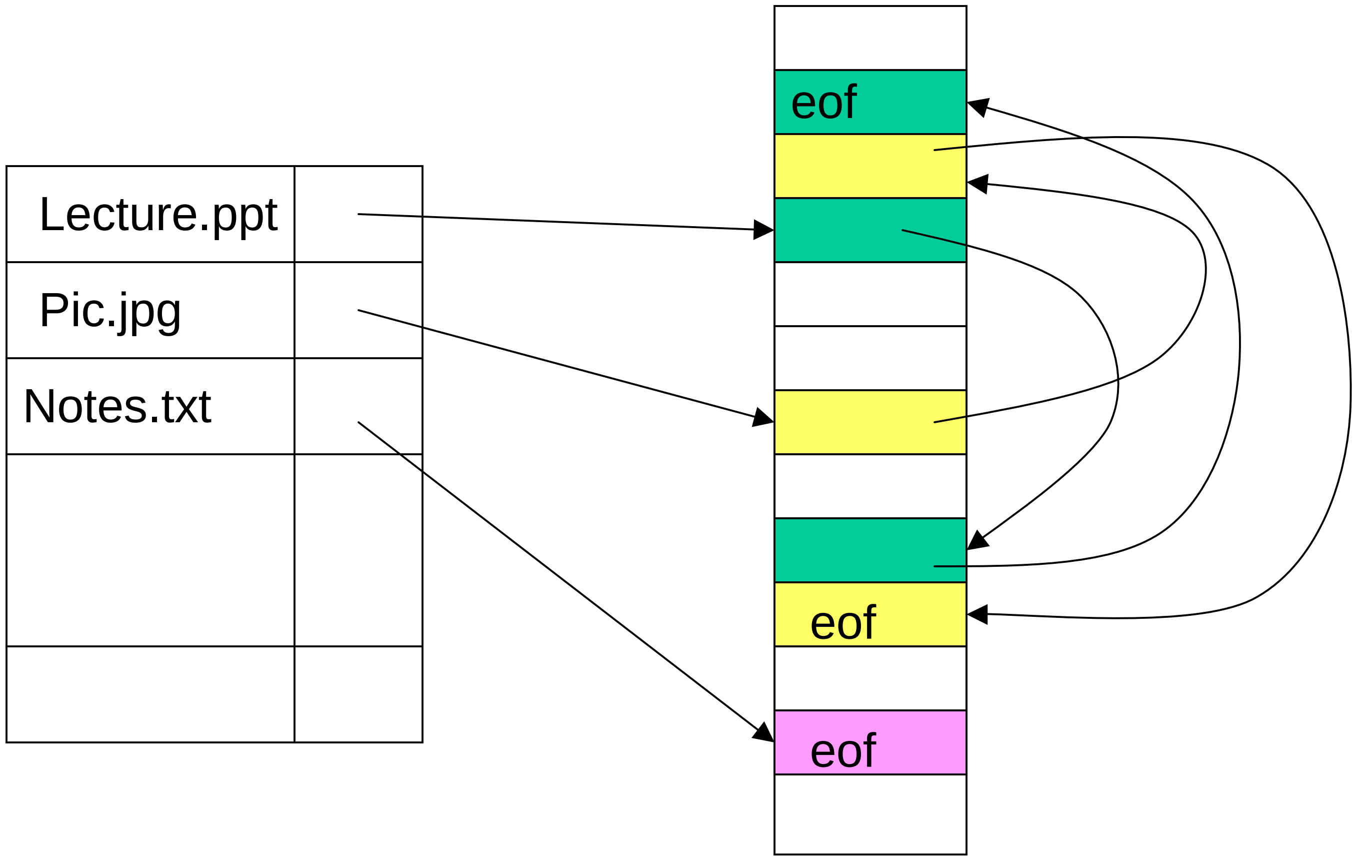
Outline for Today

- Objective
  - Continue review of basic file system material
- Administrative
  - Rest of term plan in outline
  - Program 1

File Structure Alternatives

- Contiguous
  - 1 block pointer, causes fragmentation, growth is a problem.
- Linked
  - each block points to next block, directory points to first, OK for sequential access
- Indexed
  - index structure required, better for random access into file.

File Allocation Table (FAT)

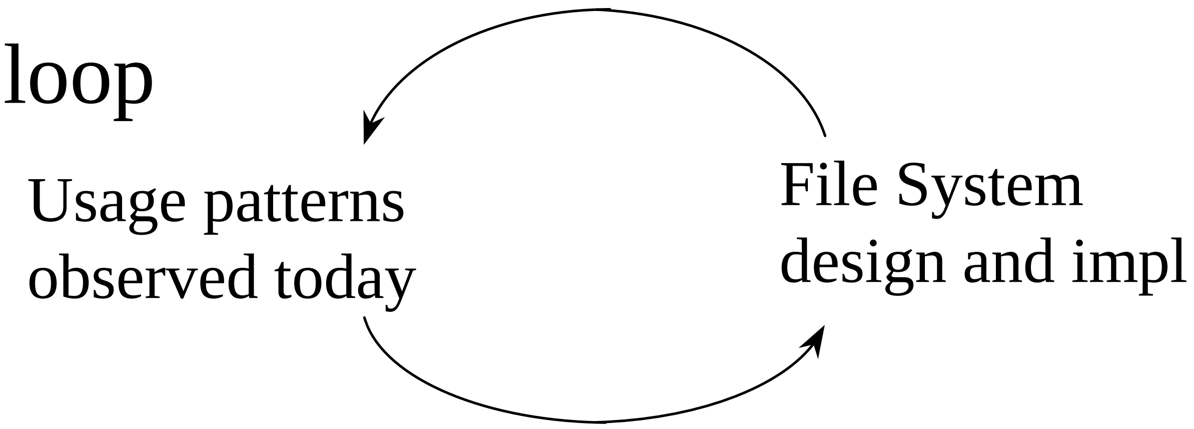


File Access Patterns

- What do users *seem* to want from the file abstraction?
- What do these usage patterns mean for file structure and implementation decisions?
  - What operations should be optimized 1st?
  - How should files be structured?
  - Is there temporal locality in file usage?
  - How long do files really live?

Know your Workload!

- File usage patterns should influence design decisions. Do things differently depending:
  - How large are most files? How long-lived? Read vs. write activity. Shared often?
  - Different levels “see” a different workload.
- Feedback loop



Generalizations from UNIX Workloads

- Standard Disclaimers that you can’t generalize…but anyway…
- Most files are small (fit into one disk block) although most bytes are transferred from longer files.
- Most opens are for read mode, most bytes transferred are by read operations
- Accesses tend to be sequential and 100%

More on Access Patterns

- There is significant reuse (re-opens) – most opens go to files repeatedly opened & quickly. Directory nodes and executables also exhibit good temporal locality.
  - Looks good for caching!
- Use of temp files is significant part of file system activity in UNIX – very limited reuse, short lifetimes (less than a minute).

What to do about long paths?

- Make long lookups cheaper – cluster inodes and data on disk to make each component resolution step somewhat cheaper
  - Immediate files – meta-data and first block of data co-located
- Collapse prefixes of paths – hash table
  - Prefix table
- “Cache it” – in this case, directory info

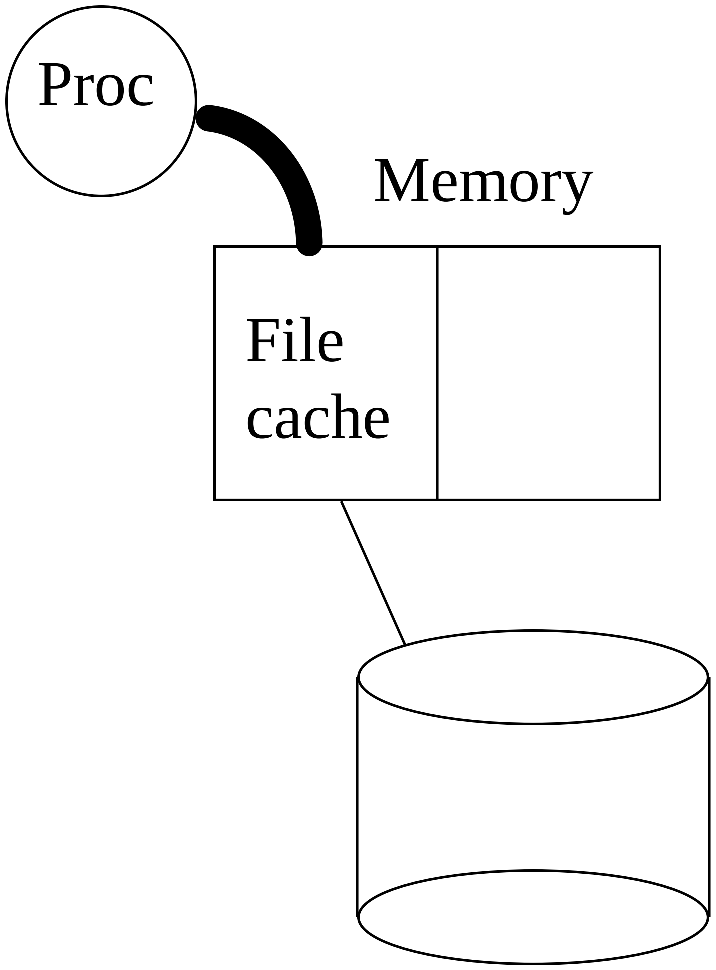
What to do about Disks?

- Disk scheduling
  - Idea is to reorder outstanding requests to minimize seeks.
- Layout on disk
  - Placement to minimize disk overhead
- Build a <u>better</u> disk (or substitute)
  - Example: RAID

Avoiding the Disk -- Caching

File Buffer Cache

- ***Avoid*** the disk for as many file operations as possible.
- Cache acts as a filter for the requests seen by the disk – reads served best.
- Delayed writeback will avoid going to disk at all for temp files.



Handling Updates in the File Cache

1. Blocks may be modified in memory once they have been brought into the cache.

   Modified blocks are *dirty* and must (eventually) be written back.

2. Once a block is modified in memory, the write back to disk may not be immediate (*synchronous*).

   Delayed writes absorb many small updates with one disk write.

   How long should the system hold dirty data in memory?

   Asynchronous writes allow overlapping of computation and disk update activity (*write-behind*).

   Do the **write** call for block ***n+1*** while transfer of block ***n*** is in progress.

Linux Page Cache

- Page Cache is the disk cache for all page-based I/O – subsumes file buffer cache.
  - All page I/O flows through page cache
- pdflush daemons – writeback to disk any dirty pages/buffers.
  - When free memory falls below threshold, wakeup daemon to reclaim free memory
    - Specified number written back
    - Free memory above threshold
  - Periodically, to prevent old data not getting written back, wakeup on timer expiration
    - Writes all pages older than specified limit.

Disk Scheduling – Seek Opt.

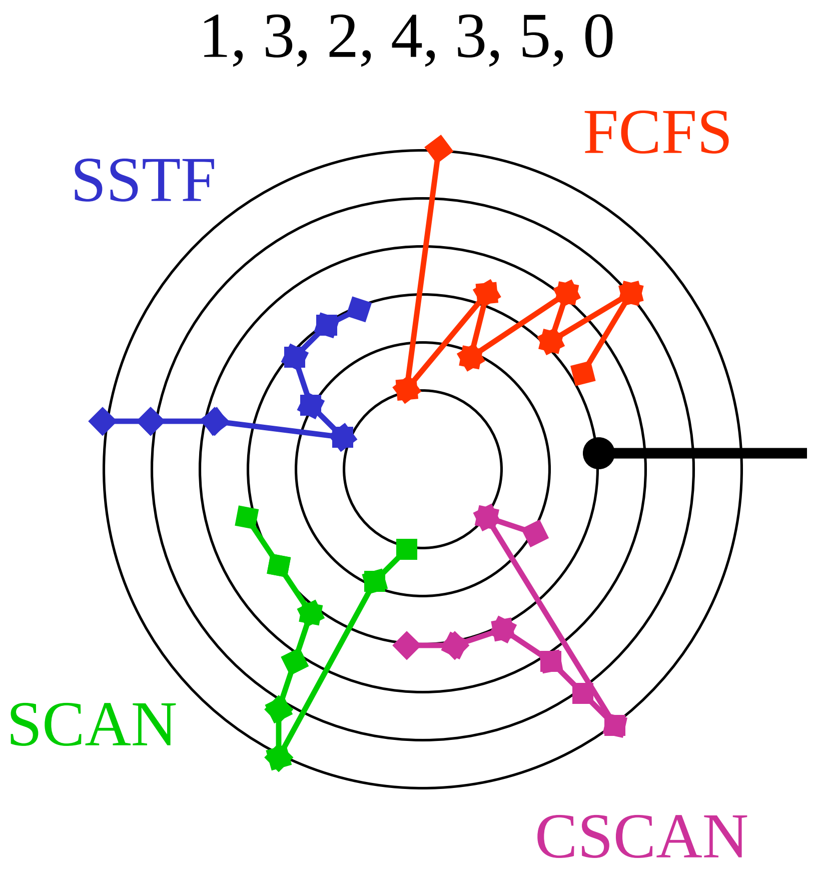
Disk Scheduling

- Assuming there are sufficient outstanding requests in request queue
- Focus is on seek time - minimizing physical movement of head.
- Simple model of seek performance

  Seek Time = startup time (e.g. 3.0 ms) +
    N (number of cylinders) *
    per-cylinder move (e.g. .04 ms/cyl)

“Textbook” Policies

- Generally use FCFS as baseline for comparison
- Shortest Seek First (SSTF) -closest
  - danger of starvation
- Elevator (SCAN) - sweep in one direction, turn around when no requests beyond
  - handle case of constant arrivals at same position
- C-SCAN - sweep in only one direction, return to 0
  - less variation in response



Linux Disk Schedulers

- Linus Elevator
  - Merging and sorting: when new request comes in
    - Merge with any enqueued request for adjacent sector
    - If any request is too old, put new request at end of queue
    - Sort by sector location in queue (between existing requests)
    - Otherwise at end
- Deadline – each request placed on 2 of 3 queues
  - sector-wise – as above
  - read FIFO and write FIFO – whenever expiration time exceeded, service from here
- Anticipatory
  - Hang around waiting for subsequent request just a bit

Disk Layout

Layout on Disk

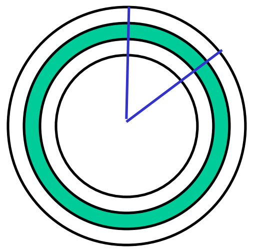
- Can address both seek and rotational latency
- Cluster related things together
  (e.g. an inode and its data, inodes in same directory (ls command), data blocks of multi-block file, files in same directory)
- Sub-block allocation to reduce fragmentation for small files
- Log-Structure File Systems

The Problem of Disk Layout

- The level of indirection in the file block maps allows flexibility in file layout.
    - “File system design is 99% block allocation.” [McVoy]
- Competing goals for block allocation:
    - *allocation cost*
    - *bandwidth* for high-volume transfers
    - *efficient directory operations*
- **Goal:** reduce disk arm movement and seek overhead.
    - metric of merit: *bandwidth utilization*

FFS Cylinder Groups

- FFS defines *cylinder groups* as the unit of disk locality, and it factors locality into allocation choices.
  - typical: thousands of cylinders, dozens of groups
  - <u>Strategy</u>: place "related" data blocks in the same cylinder group whenever possible.
    - seek latency is proportional to seek distance
  - Smear large files across groups:
    - Place a run of contiguous blocks in each group.
  - Reserve inode blocks in each cylinder group.
    - This allows inodes to be allocated close to their directory entries and close to their data blocks (for small files).



FFS Allocation Policies

1. Allocate file inodes close to their containing directories.
   - For *mkdir*, select a cylinder group with a more-than-average number of free inodes.
   - For *creat*, place inode in the same group as the parent.
2. Concentrate related file data blocks in cylinder groups.
      - Most files are read and written sequentially.
   - Place initial blocks of a file in the same group as its inode.
      - How should we handle directory blocks?
   - Place adjacent logical blocks in the same cylinder group.
      - Logical block ***n+1*** goes in the same group as block ***n.***
      - Switch to a different group for each indirect block.

Allocating a Block

1. Try to allocate the rotationally optimal physical block after the previous logical block in the file.

   Skip *rotdelay* physical blocks between each logical block.

   (rotdelay is 0 on track-caching disk controllers.)

2. If not available, find another block a nearby rotational position in the same cylinder group

   We'll need a short seek, but we won't wait for the rotation.

   If not available, pick any other block in the cylinder group.

3. If the cylinder group is full, or we're crossing to a new indirect block, go find a new cylinder group.

   Pick a block at the beginning of a run of free blocks.

Clustering in FFS

- *Clustering* improves bandwidth utilization for large files read and written sequentially.
    - Allocate clumps/clusters/runs of blocks contiguously; read/write the entire clump in one operation with at most one seek.
  - Typical cluster sizes: 32KB to 128KB.
- FFS can allocate contiguous runs of blocks "most of the time" on disks with sufficient free space.
  - This (usually) occurs as a side effect of setting *rotdelay* = 0.
    - Newer versions may relocate to clusters of contiguous storage if the initial allocation did not succeed in placing them well.
  - Must modify buffer cache to group buffers together and read/write in contiguous clusters.

Effect of Clustering

Access time = seek time + rotational delay + transfer time

average seek time = 2 ms for an intra-cylinder group seek, let's say
rotational delay = 8 milliseconds for full rotation at 7200 RPM: average
delay = 4 ms
transfer time = 1 millisecond for an 8KB block at 8 MB/s

8 KB blocks deliver about 15% of disk bandwidth.
64KB blocks/clusters deliver about 50% of disk bandwidth.
128KB blocks/clusters deliver about 70% of disk bandwidth.

Actual performance will likely be better with good disk layout, since most seek/rotate delays to read the next block/cluster will be "better than average".

Log-Structured File Systems

- Assumption: Cache is effectively filtering out reads so we should optimize for writes
- Basic Idea: manage disk as an append-only ***log*** (subsequent writes involve minimal head movement)
- Data and meta-data (mixed) accumulated in large segments and written contiguously
- Reads work as in UNIX - once inode is found, data blocks located via index.
- Cleaning an issue - to produce contiguous free space, correcting fragmentation developing over time.
- Claim: LFS can use 70% of disk bandwidth for writing while Unix FFS can use only 5-10% typically because of seeks.

LFS logs

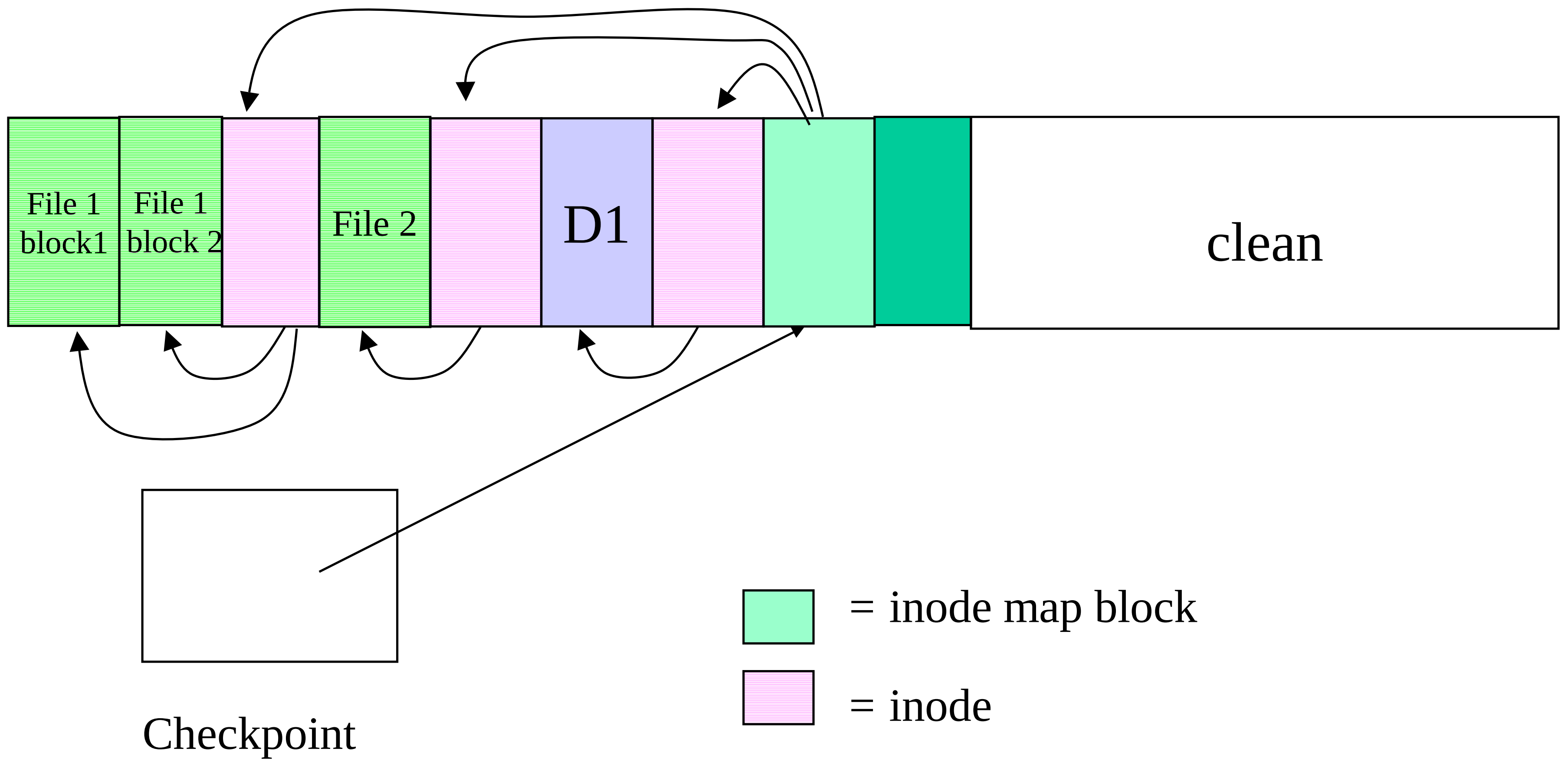
In LFS, *all* block and metadata allocation is log-based.

- LFS views the disk as “one big log” (logically).
- *All* writes are clustered and sequential/contiguous.
  - Intermingles metadata and blocks from different files.
- Data is laid out on disk in the order it is written.
- No-overwrite allocation policy: if an old block or inode is modified, write it to a new location at the *tail* of the log.
- LFS uses (mostly) the same metadata structures as FFS; only the allocation scheme is different.
  - Cylinder group structures and free block maps are eliminated.
  - Inodes are found by indirecting through a new map

LFS Data Structures on Disk

- Inode – in log, same as FFS
- Inode map – in log, locates position of inode, version, time of last access
- Segment summary – in log, identifies contents of segment (file#, offset for each block in segment)
- Segment usage table – in log, counts live bytes in segment and last write time
- Checkpoint region – fixed location on disk, locates blocks of inode map, identifies last checkpoint in log.
- Directory change log – in log, records directory operations to maintain consistency of ref counts in inodes

Structure of the Log



🟩 = inode map block

▨ = inode

▨ = directory node

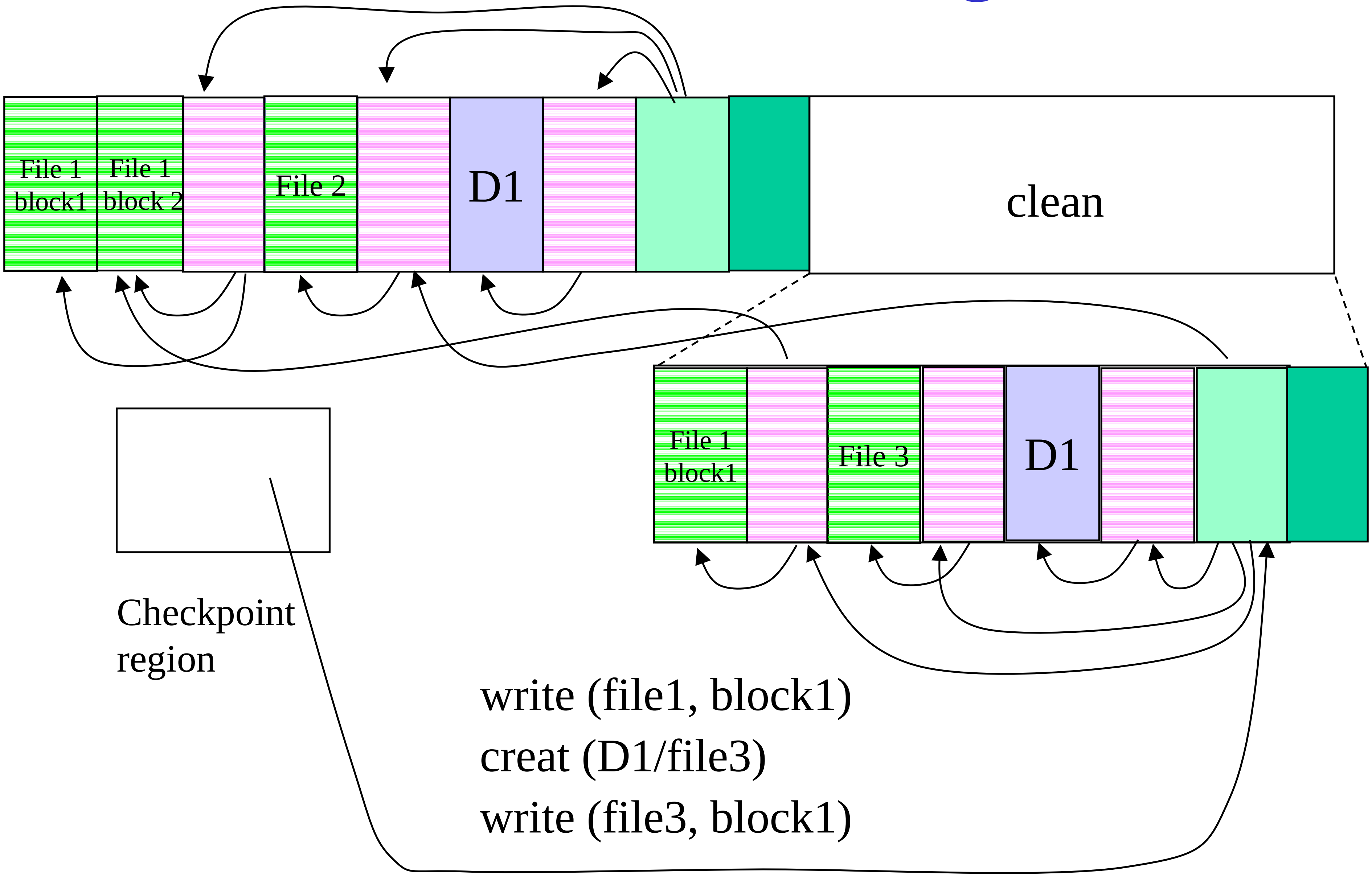
▨ = data block

🟩 = segment summary, usage, dirlog

Writing the Log in LFS

1. LFS "saves up" dirty blocks and dirty inodes until it has a full *segment* (e.g., 1 MB).
   - Dirty inodes are grouped into block-sized clumps.
   - Dirty blocks are sorted by *(file, logical block number).*
   - Each log segment includes summary info and a checksum.
2. LFS writes each log segment in a single burst, with at most one seek.
   - Find a free segment "slot" on the disk, and write it.
   - Store a back pointer to the previous segment.
     - Logically the log is sequential, but physically it consists of a chain of segments, each large enough to amortize seek overhead.

Growth of the Log

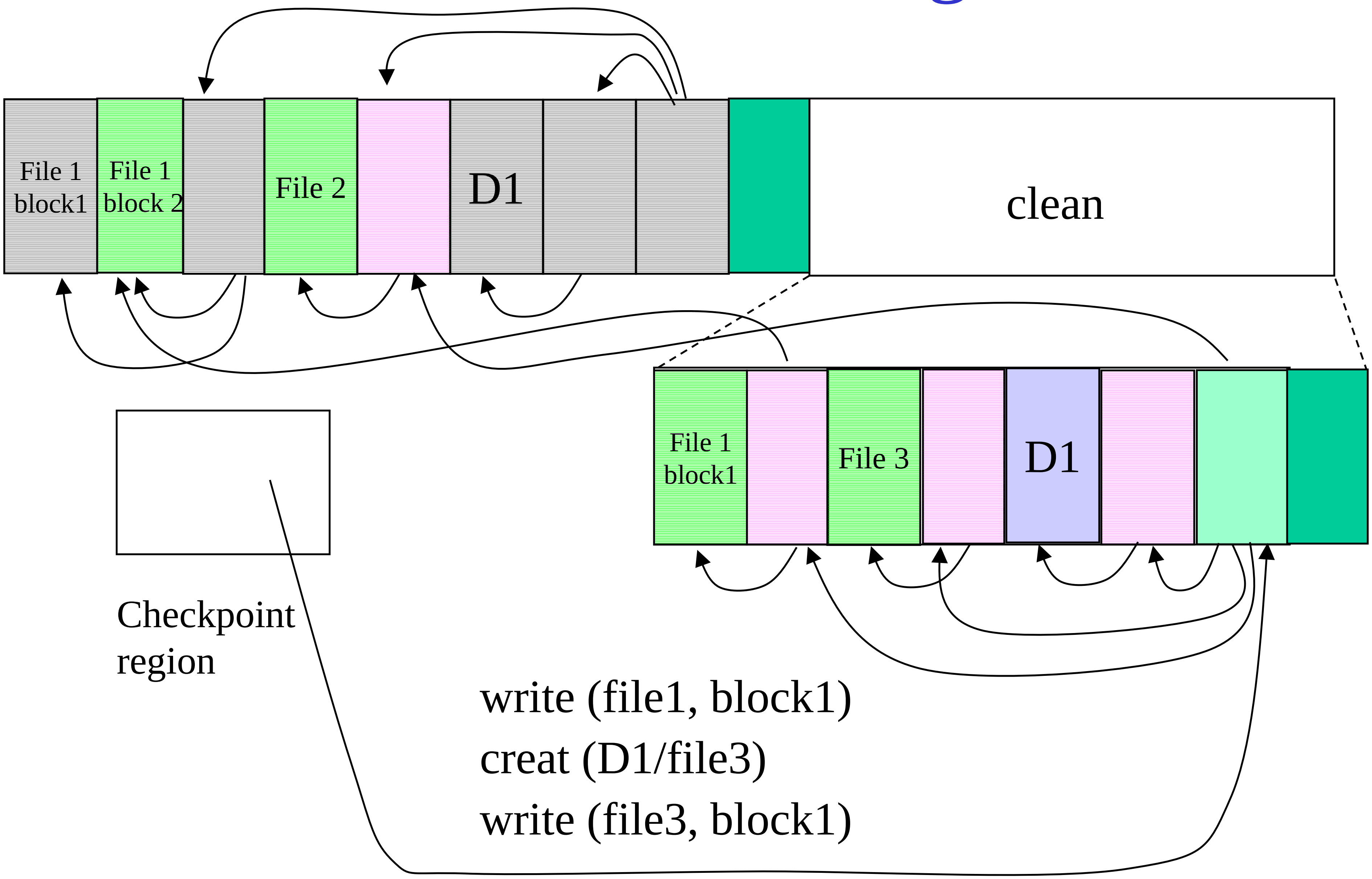


write (file1, block1)

creat (D1/file3)

write (file3, block1)

Death in the Log



write (file1, block1)
creat (D1/file3)
write (file3, block1)

Writing the Log: the Rest of the Story

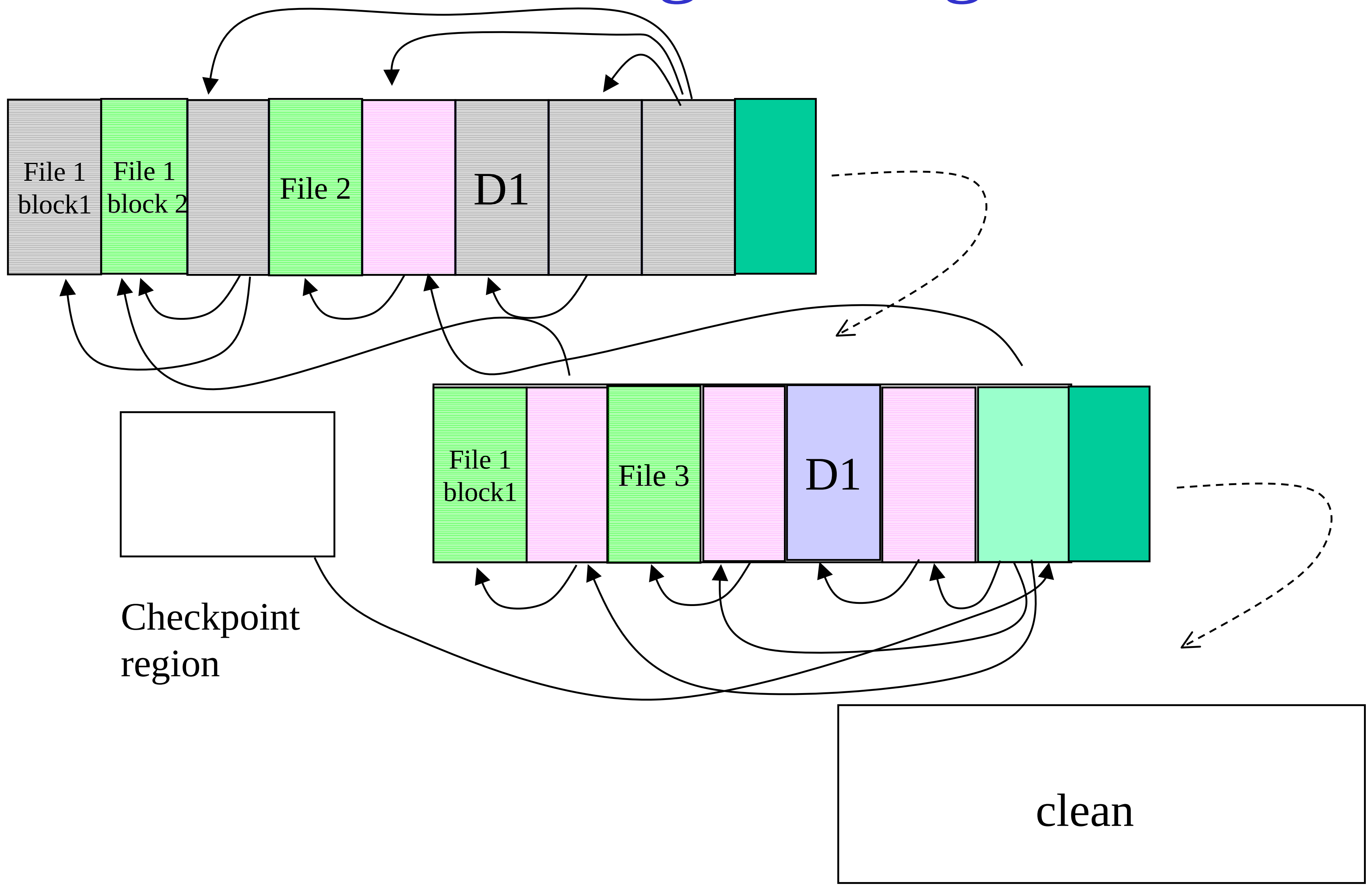
1. LFS cannot always delay writes long enough to accumulate a full segment; sometimes it must push a *partial segment*.
   - fsync, update daemon, NFS server, etc.
   - Directory operations are synchronous in FFS, and some must be in LFS as well to preserve failure semantics and ordering.
2. LFS allocation and write policies affect the buffer cache, which is supposed to be filesystem-independent.
   - Pin (*lock*) dirty blocks until the segment is written; dirty blocks cannot be recycled off the free chain as before.
   - Endow *indirect blocks with permanent logical block numbers suitable for hashing in the buffer cache.

Cleaning in LFS

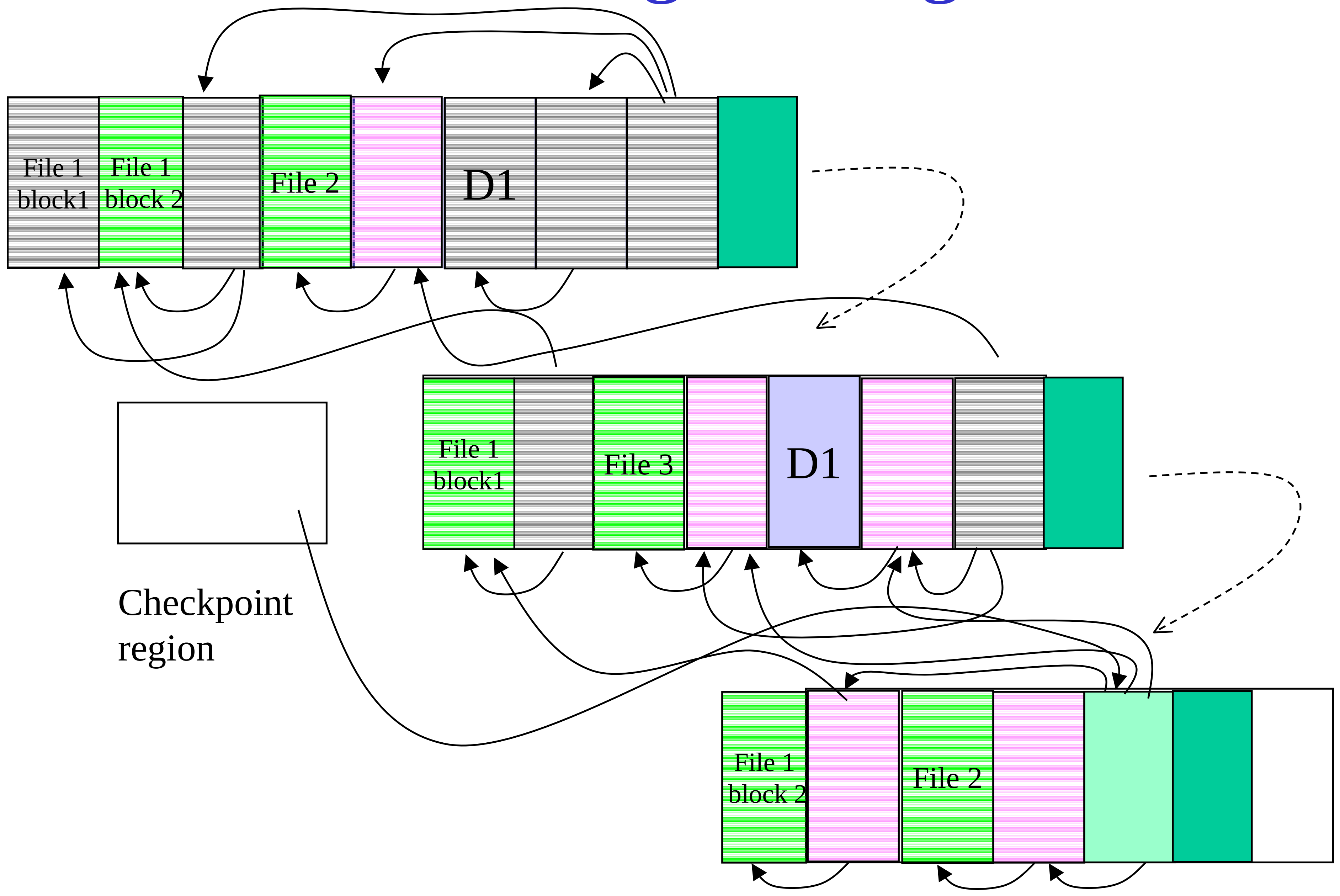
What does LFS do when the disk fills up?

1. As the log is written, blocks and inodes written earlier in time are superseded (“killed”) by versions written later.
   - files are overwritten or modified; inodes are updated
   - when files are removed, blocks and inodes are deallocated
2. A cleaner daemon compacts remaining live data to free up large hunks of free space suitable for writing segments.
   - look for segments with little remaining live data
     - benefit/cost analysis to choose segments
   - write remaining live data to the log tail
   - can consume a significant share of bandwidth, and there are lots of cost/benefit heuristics involved.

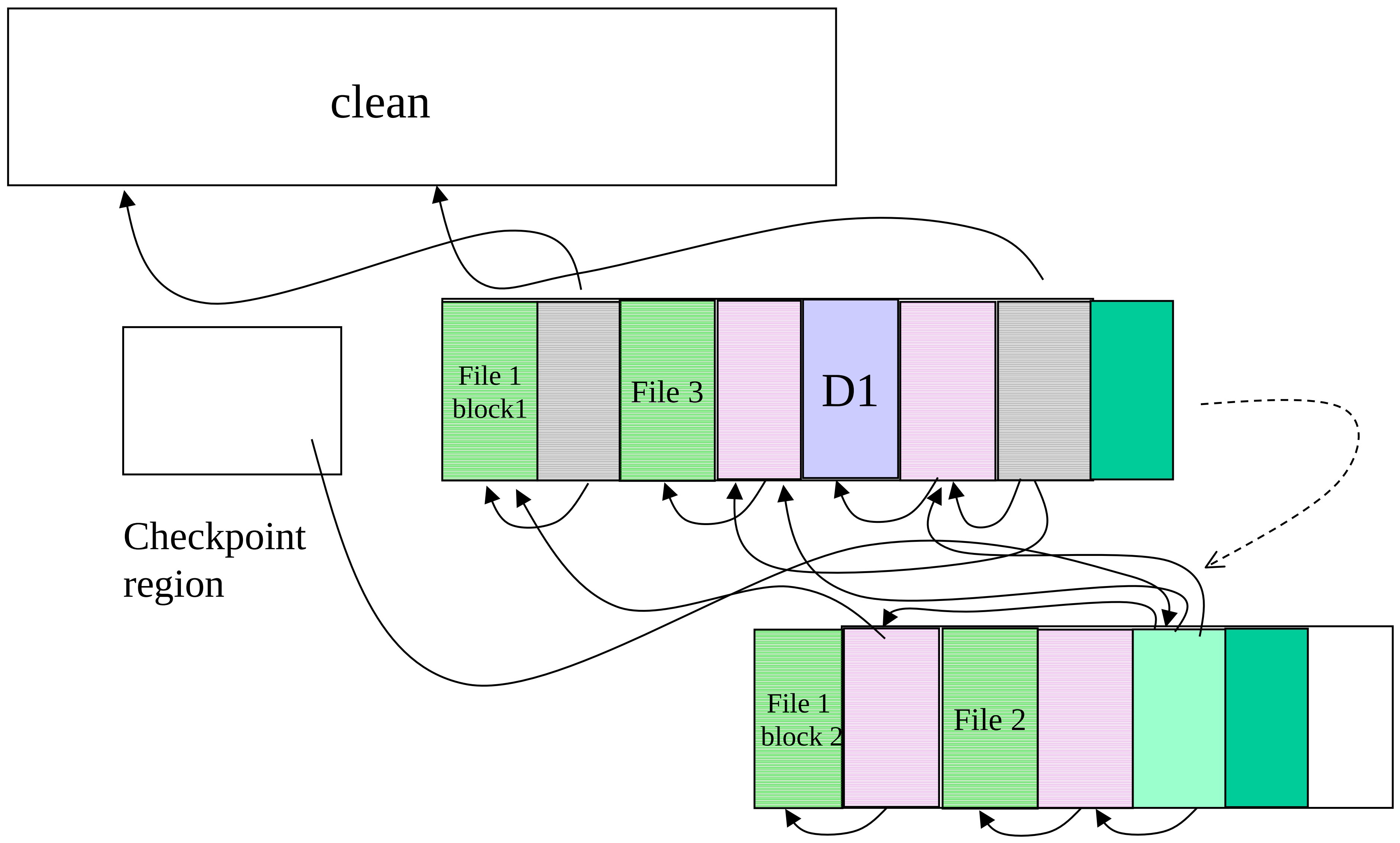
Cleaning the Log



Cleaning the Log



Cleaning the Log



Cleaning Issues

- Must be able to identify which blocks are live
- Must be able to identify the file to which each block belongs in order to update inode to new location
- Segment Summary block contains this info
  - File contents associated with uid (version # and inode #)
  - Inode entries contain version # (incr. on truncate)
  - Compare to see if inode points to block under consideration

Policies

- When cleaner cleans – threshold based
- How much – 10s at a time until threshold reached
- Which segments
  - Most fragmented segment is not best choice.
  - Value of free space in segment depends on stability of live data (approx. age)
  - Cost / benefit analysis

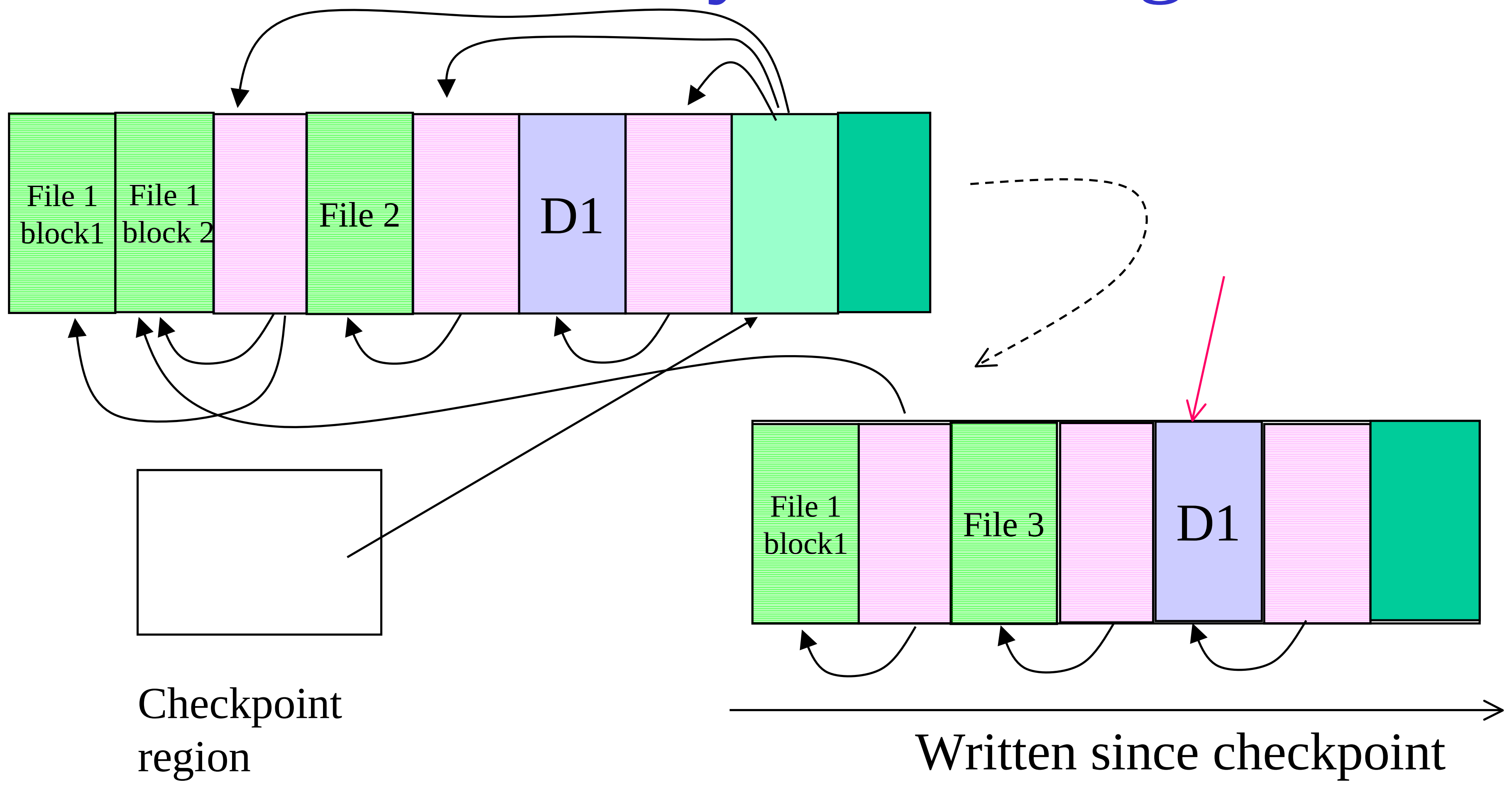
    Benefit = free space available (1-u) * age of youngest block

    Cost = cost to read segment + cost to move live data
  - Segment usage table supports this
- How to group live blocks

Recovering Disk Contents

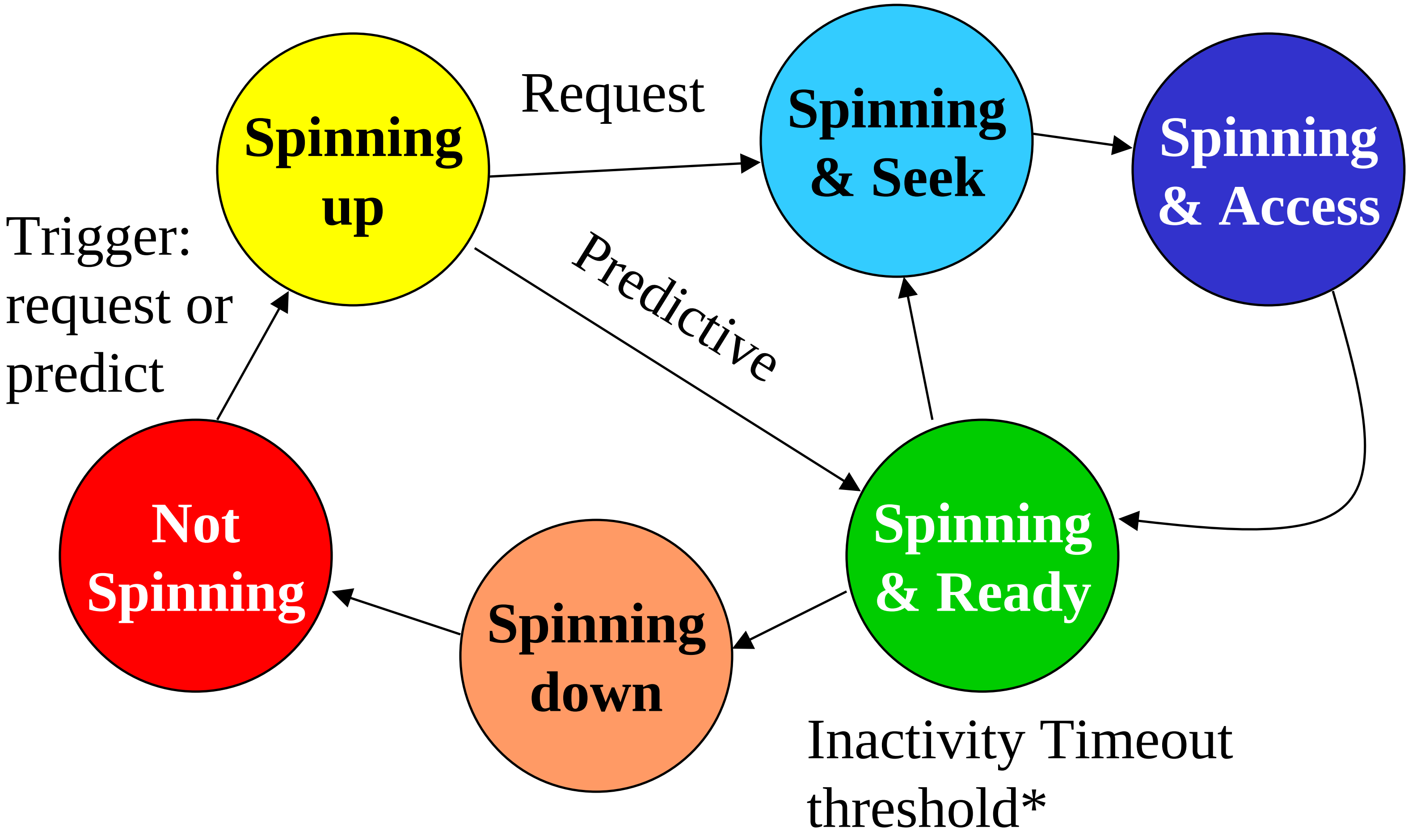
- Checkpoints – define consistent states
  - Position in log where all data structures are consistent
  - Checkpoint region (fixed location) – contains the addresses of all blocks of inode map and segment usage table, ptr to last segment written
    - Actually 2 that alternate in case a crash occurs while writing checkpoint region data
- Roll-forward – to recover beyond last checkpoint
  - Uses Segment summary blocks at end of log – if we find new inodes, update inode map found from checkpoint
  - Adjust utilizations in segment usage table
  - Restore consistency in ref counts within inodes and directory entries pointing to those inodes using Directory operation log (like an intentions list)

Recovery of the Log



Disk Power Management

Spin-down Disk Model



Reducing Energy Consumption

$$\text{Energy} = \sum_{i\,\varepsilon\,\text{power states}} \text{Power}_i \times \text{Time}_i$$

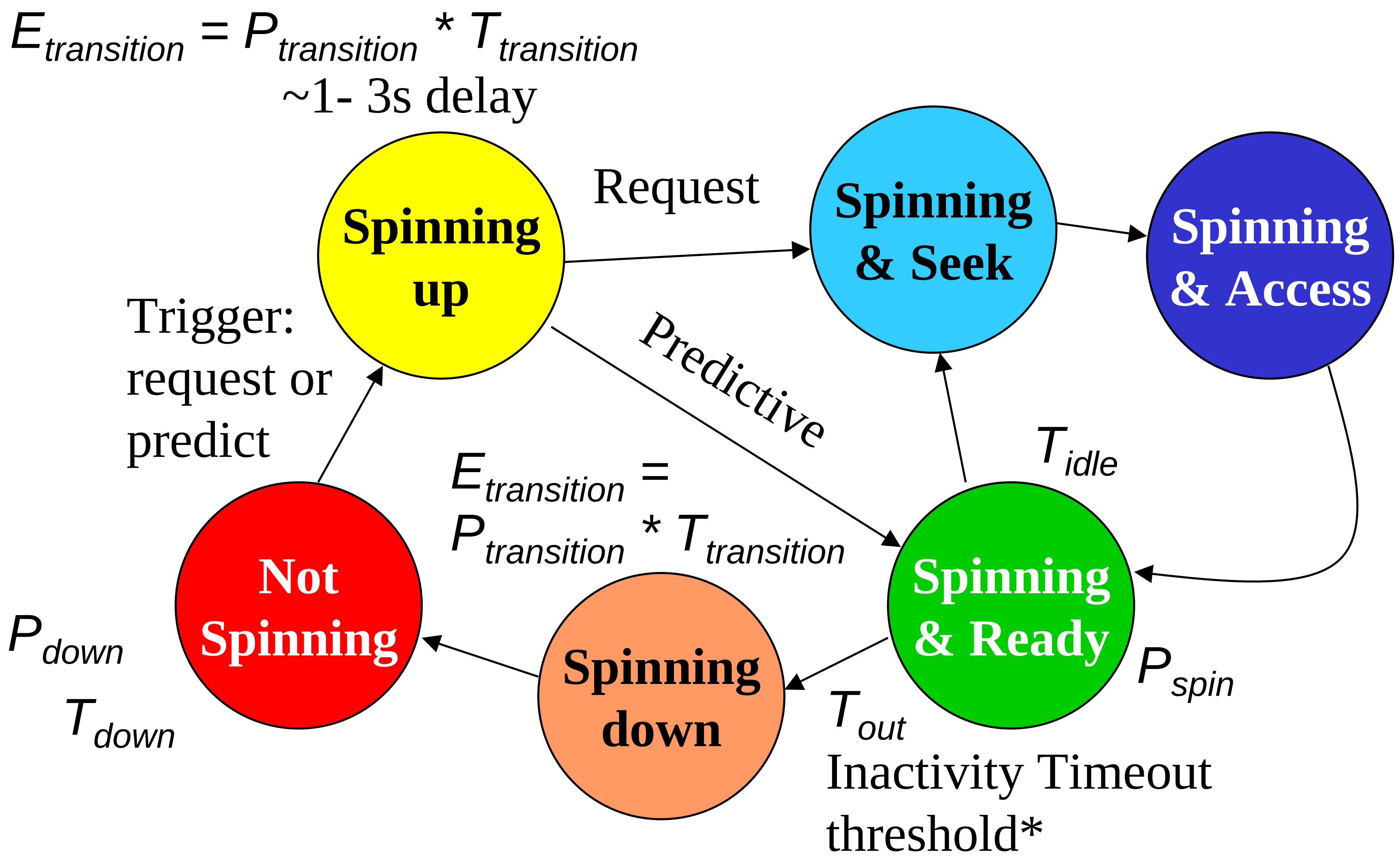
To reduce energy used for task:

- Reduce power cost of power state *I* through better technology.
- Reduce time spent in the higher cost power states.
- Amortize transition states (spinning up or down) if significant.

$$P_{down} * T_{down} + 2 * E_{transition} + P_{spin} * T_{out} < P_{spin} * T_{idle}$$

$$T_{down} = T_{idle} - (T_{transition} + T_{out})$$

Spin-down Disk Model



Power Specs

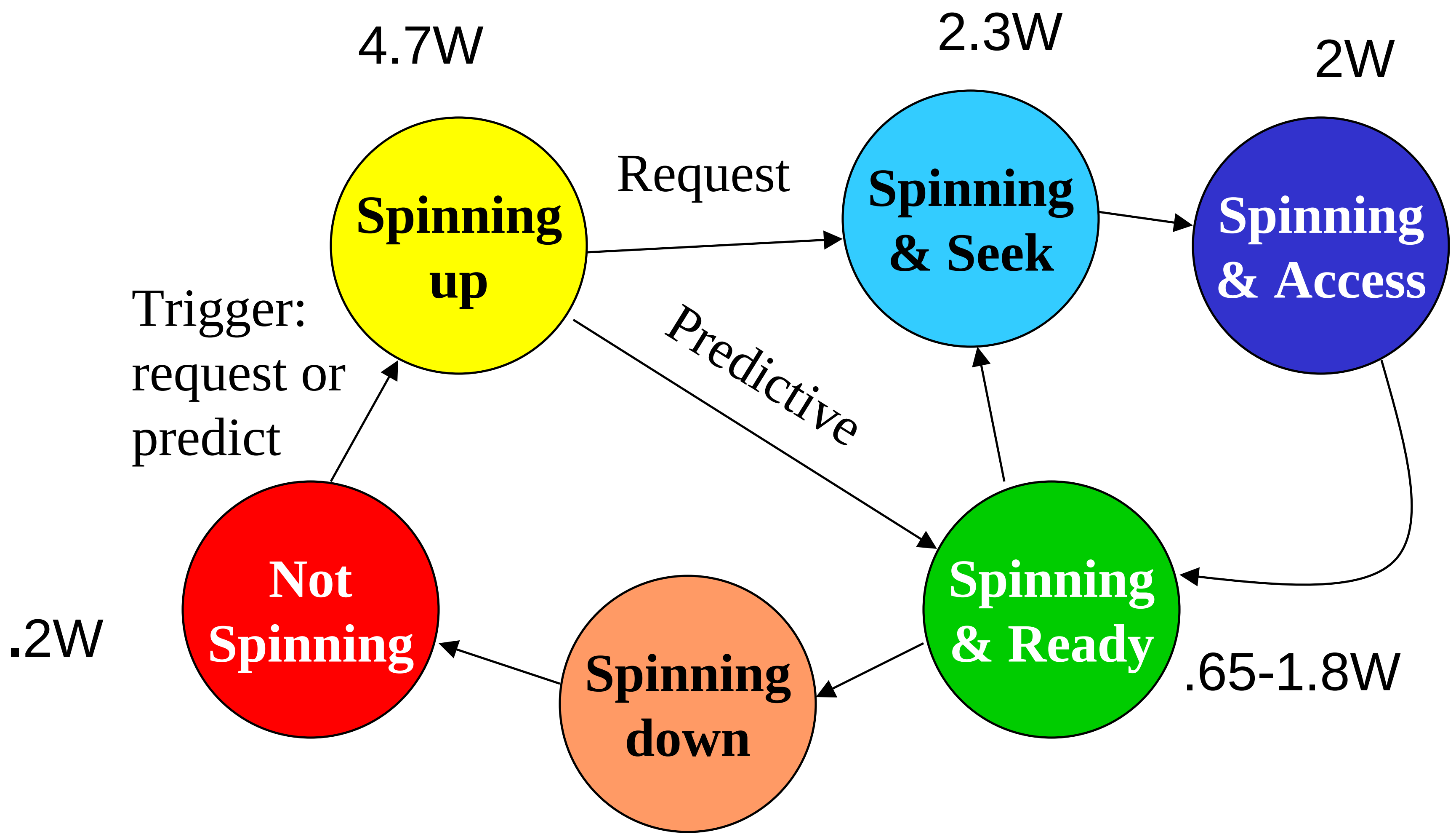
IBM Microdrive (1inch)

- writing 300mA (3.3V) 1W
- standby 65mA (3.3V) .2W

IBM TravelStar (2.5inch)

- read/write 2W
- spinning 1.8W
- low power idle .65W
- standby .25W
- sleep .1W
- startup 4.7 W
- seek 2.3W

Spin-down Disk Model



Spin-Down Policies

- Fixed Thresholds
  - T_{out} = spin-down cost s.t. $2*E_{transition} = P_{spin}*T_{out}$
- Adaptive Thresholds: T_{out} = f (recent accesses)
  - Exploit burstiness in T_{idle}
- Minimizing Bumps (user annoyance/latency)
  - Predictive spin-ups
- Changing access patterns (making burstiness)
  - Caching
  - Prefetching

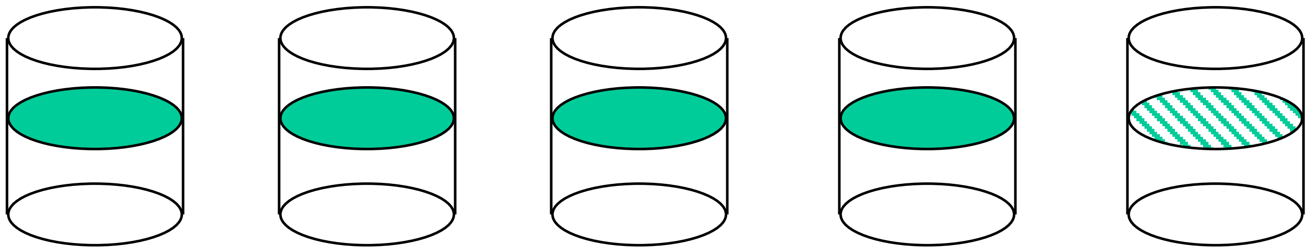
Disk Alternatives

Build a Better Disk?

- “Better” has typically meant density to disk manufacturers - bigger disks are better.
- I/O Bottleneck - a speed disparity caused by processors getting faster more quickly
- One idea is to use parallelism of multiple disks
  - ***Striping*** data across disks
  - Reliability issues - introduce redundancy

RAID

Redundant Array of Inexpensive Disks

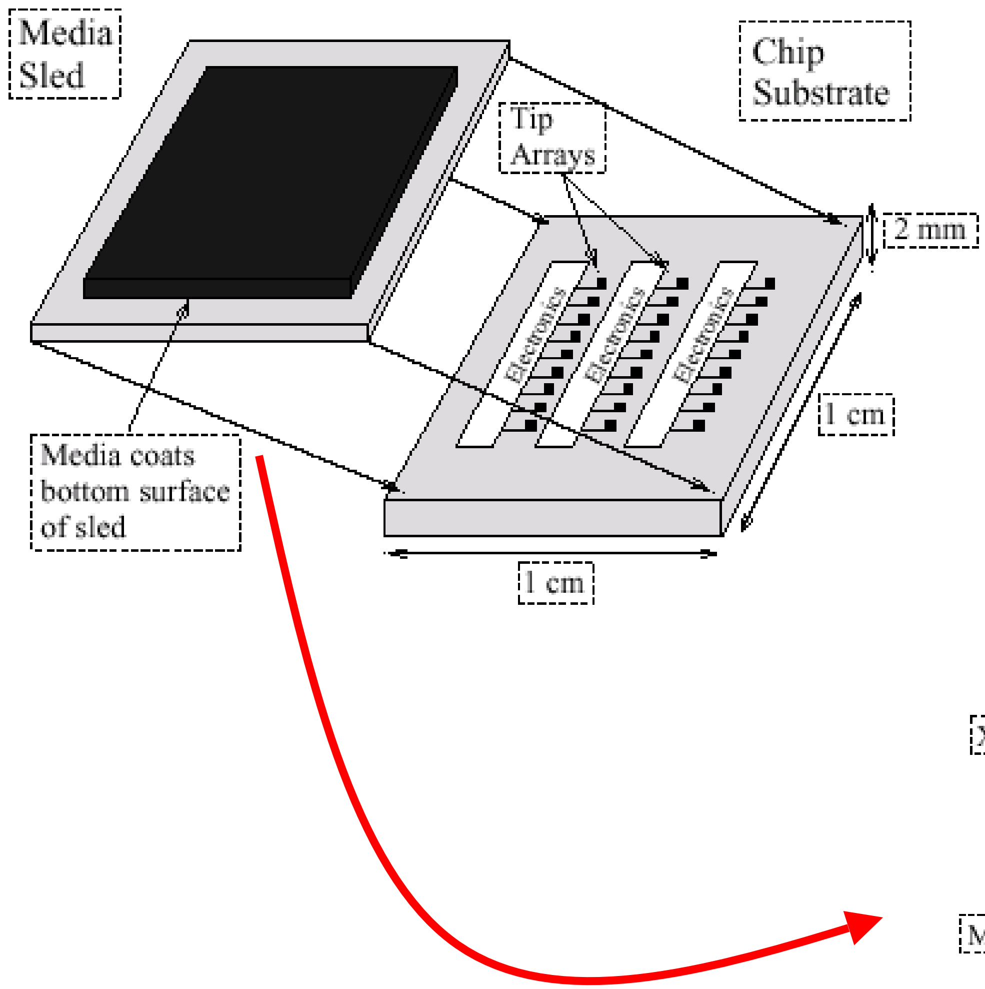


Striped Data

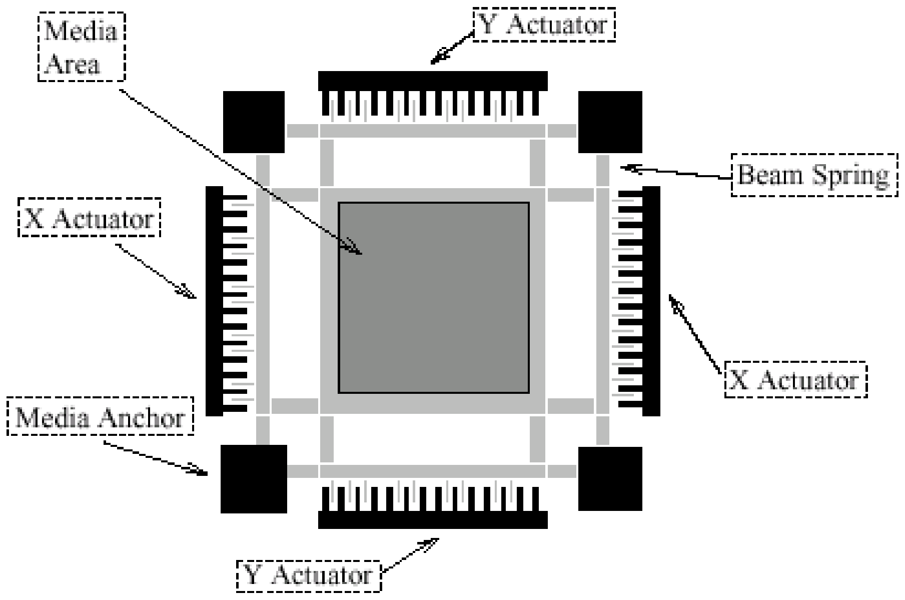
Parity
Disk

(RAID Levels 2 and 3)

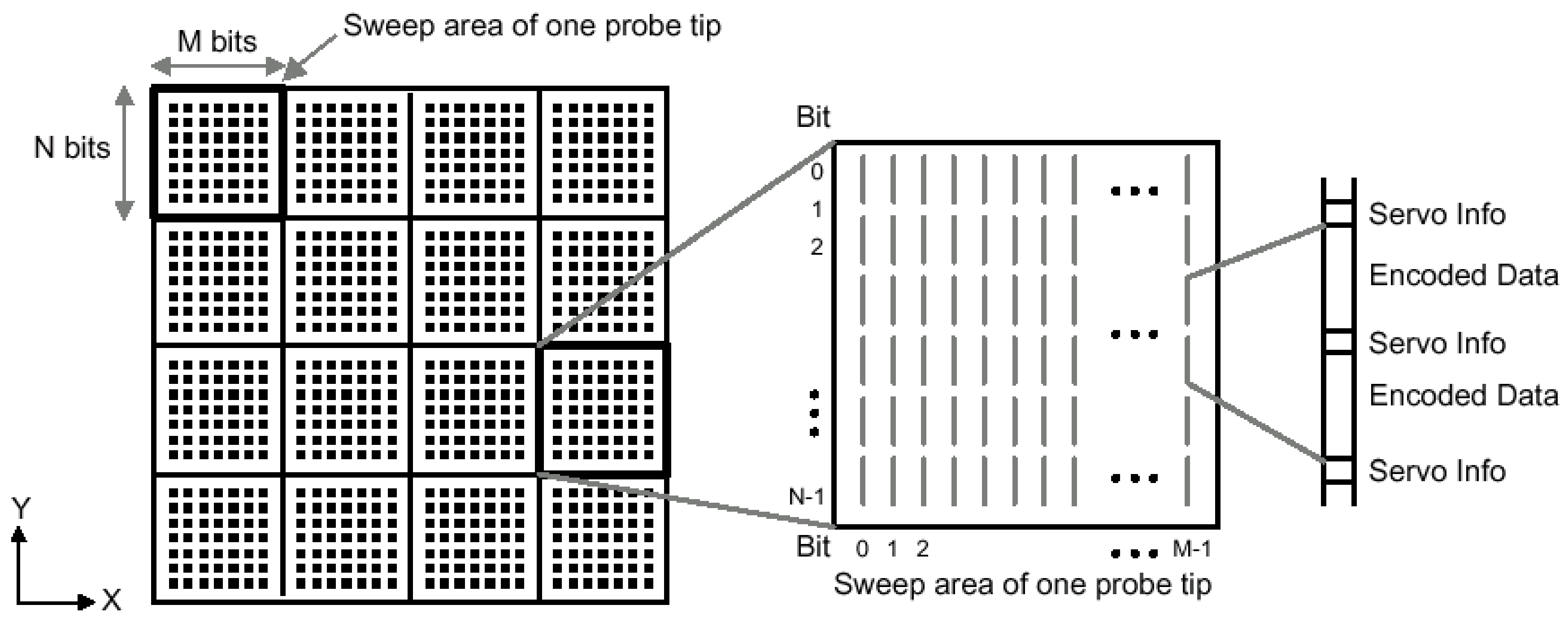
MEMS-based Storage



- Settling time after X seek
- Spring factor - non-uniform over sled positions
- Turnaround time



Data on Media Sled



Distributed File Systems

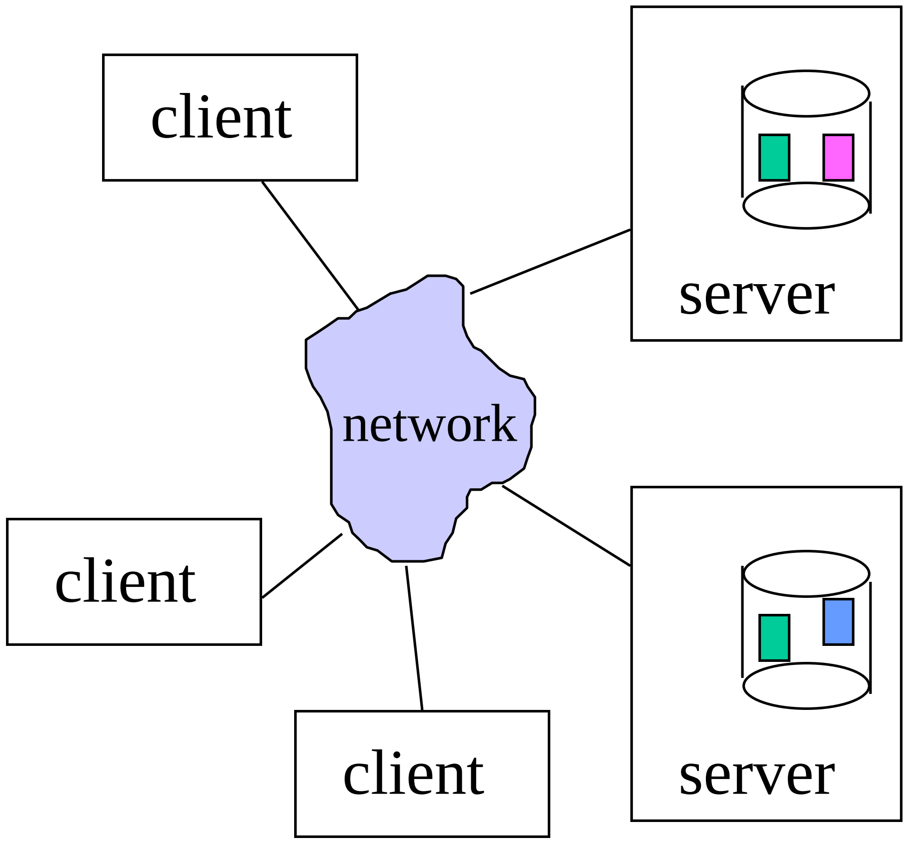
Remote Storage

&

Caching

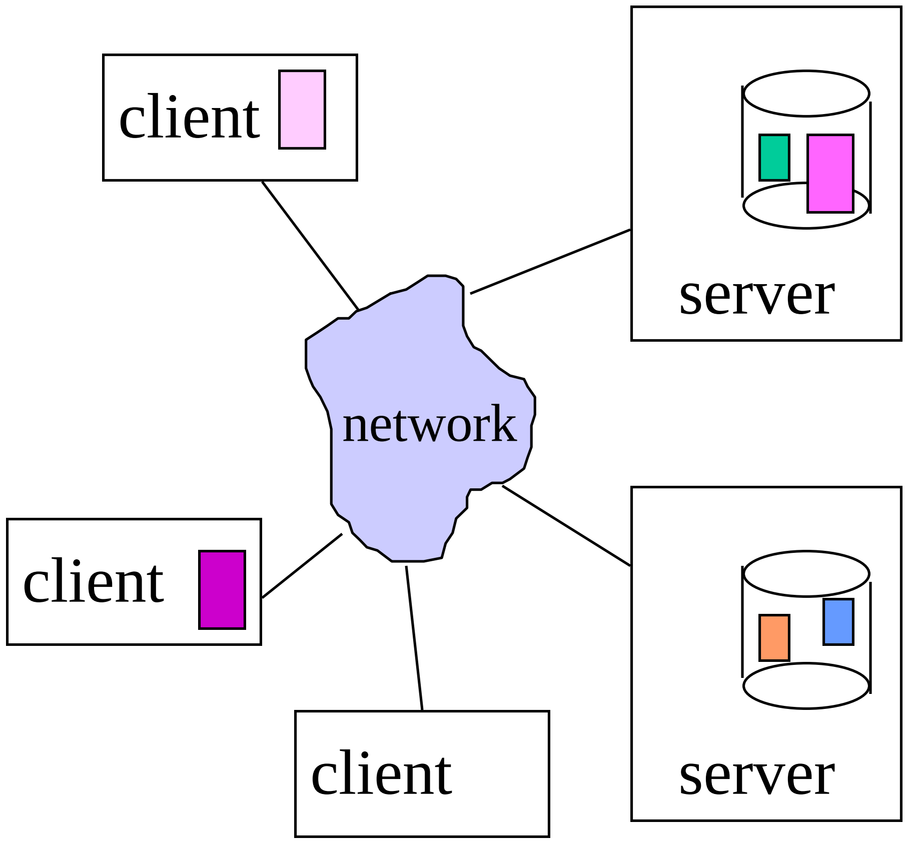
Distributed File Systems

- Naming
  - Location transparency/ independence
- Caching
  - Consistency
- Replication
  - Availability and updates



Cache Consistency

- Location of cache on client - disk or memory
- Update policy
  - write through
  - delayed writeback
  - write-on-close
- Consistency
  - Client does validity check, contacting server
  - Server call-backs

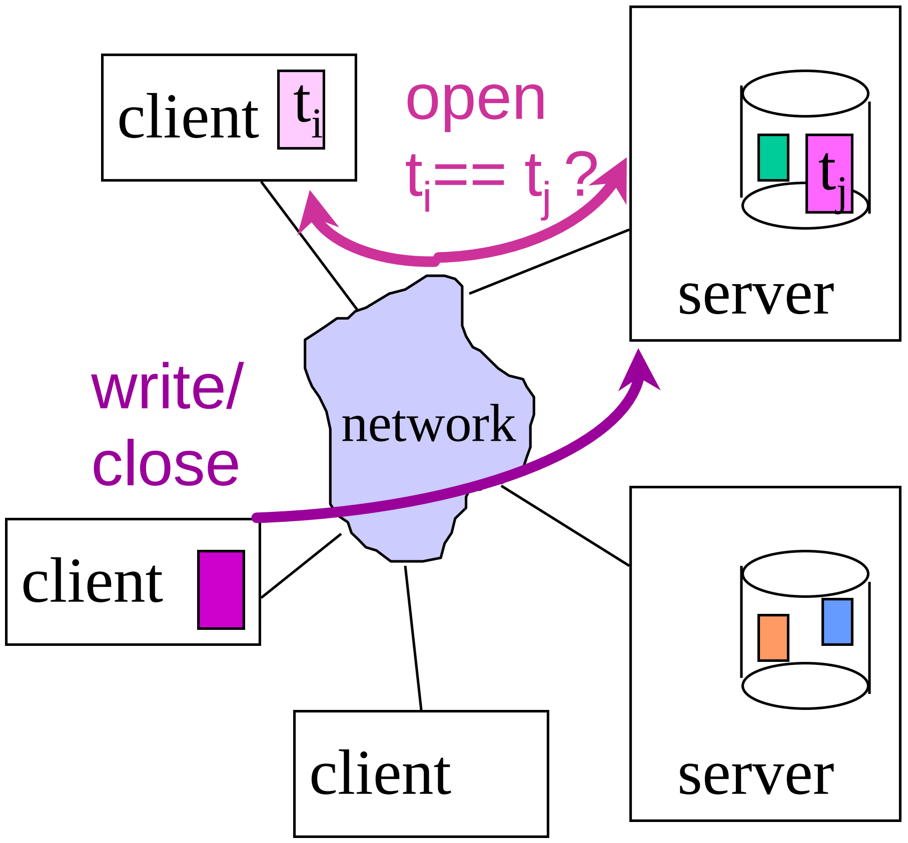


File Cache Consistency

- Caching is a key technique in distributed systems.
    - The *cache consistency problem*: cached data may become *stale* if cached data is updated elsewhere in the network.
- Solutions:
    - *Timestamp invalidation* (NFS).
        - Timestamp each cache entry, and periodically query the server: “has this file changed since time ***t***?”; invalidate cache if stale.
    - *Callback invalidation* (AFS).
        - Request notification (callback) from the server if the file changes; invalidate cache on callback.
    - *Leases* (NQ-NFS) [Gray&Cheriton89]

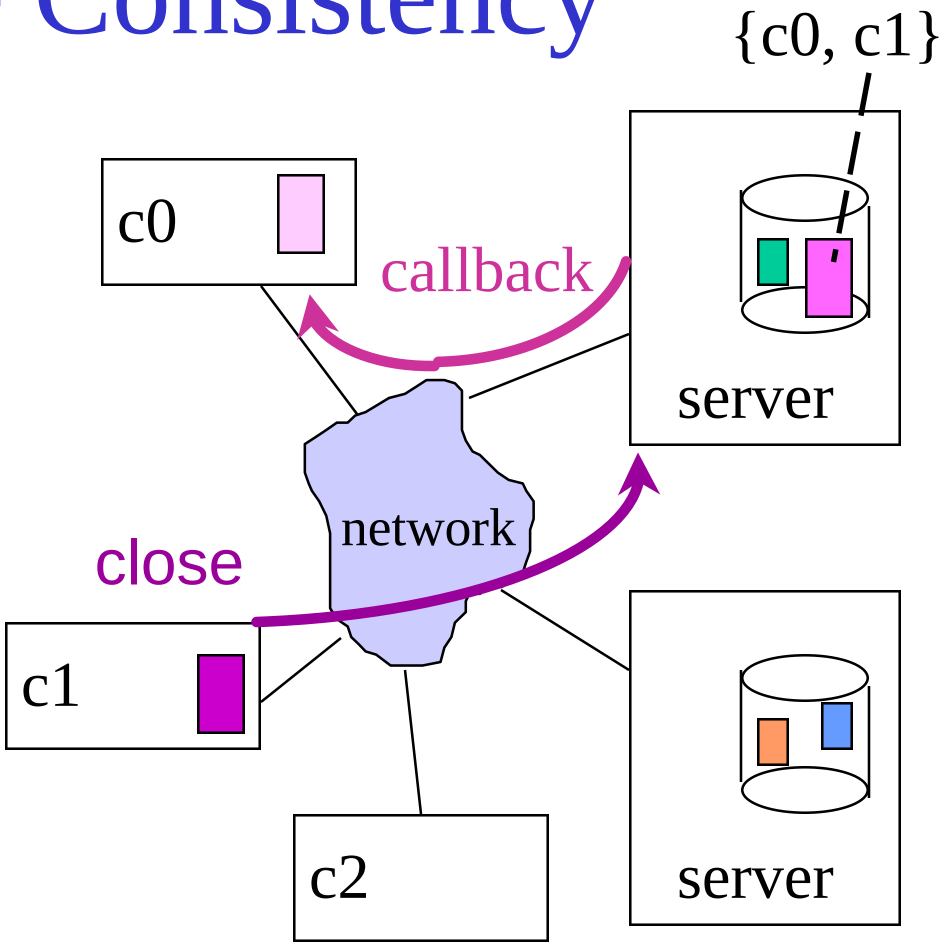
Sun NFS Cache Consistency

- Server is ***stateless***
- Requests are self-contained.
- ***Blocks*** are transferred and cached in memory.
- Timestamp of last known mod kept with cached file, compared with "true" timestamp at server on Open.
  (Good for an interval)
- Updates delayed but flushed before Close ends.



AFS Cache Consistency

- Server keeps state of all clients holding copies (copy set)
- ***Callbacks*** when cached data are about to become stale
- Large units (whole files or 64K portions)
- Updates propagated upon close
- Cache on local disk –> memory



NQ-NFS Leases

- In NQ-NFS, a client obtains a *lease* on the file that permits the client's desired read/write activity.
  - "A lease is a ticket permitting an activity; the lease is valid until some expiration time." - temporary statefulness
  - A *read-caching lease* allows the client to cache clean data.
    - **Guarantee**: no other client is modifying the file.
  - A *write-caching lease* allows the client to buffer modified data for the file. Must push data before expiration
    - **Guarantee**: no other client has the file cached.
- Leases may be revoked by the server if another client requests a conflicting operation (server sends *eviction notice*). Push in *write_slack* period.

Explicit First-class Replication

- File name maps to set of replicas, one of which will be used to satisfy request
  - Goal: availability
- Update strategy
  - Atomic updates - all or none
  - Primary copy approach
  - Voting schemes
  - Optimistic, then detection of conflicts

Optimistic vs. Pessimistic

- High availability Conflicting updates are the potential problem - requiring detection and resolution.

- Avoids conflicts by holding of shared or exclusive locks.
- How to arrange when disconnection is involuntary?
- Leases [Gray, SOSP89] puts a time-bound on locks but what about expiration?