Agenda

File System Design: Additional considerations

1. Journaling file systems
2. Block Size
3. Quotas
4. Backups
5. Consistency checking
6. Performance
7. Defragmentation

1. Journaling File Systems

Idea: Use logging to help recover from crashes

Example: Consider removing a file. What 3 things need to happen?

Now consider: how could a crash could leave the file system in an inconsistent state?

1. Journaling File System

Solution: Log all operations needed to achieve a task. Only clear the operations once the entire task is complete

When you boot, look at the journal. If there are any outstanding tasks, repeat the operations until complete

Journaling relies on **idempotent** operations, e.g. we can repeat them without harm.

Aside: Idempotent actions

Which of the following are idempotent?

- Setting a bit to 0
- Adding nodes to the end of a linked list
- Adding a value to a hashtable
- Removing a filename from a directory

2. Blocksize

The ideal blocksize is the one fits most files in a single block

- Example: in 2006, Tanenbaum found that 90.84% of all files were 64 KB or smaller
- In general, we don’t know what the ideal size will be until a computer is in use (and these will tend to evolve over time)

Trade-offs

- Smaller block size leads to less waste in a block
- Larger block size leads to better data transfer rate

3. Quotas

Problem: On a shared system, we need to prevent a user from taking all the space

Solution: Give each user a files/storage quota and keep track of memory usage by account

A **soft cap** limit can be exceeded for a set amount of time. Ignoring the soft cap for too long will lock you out of your account.

A **hard cap** cannot be exceeded. Writing files above this amount will fail.

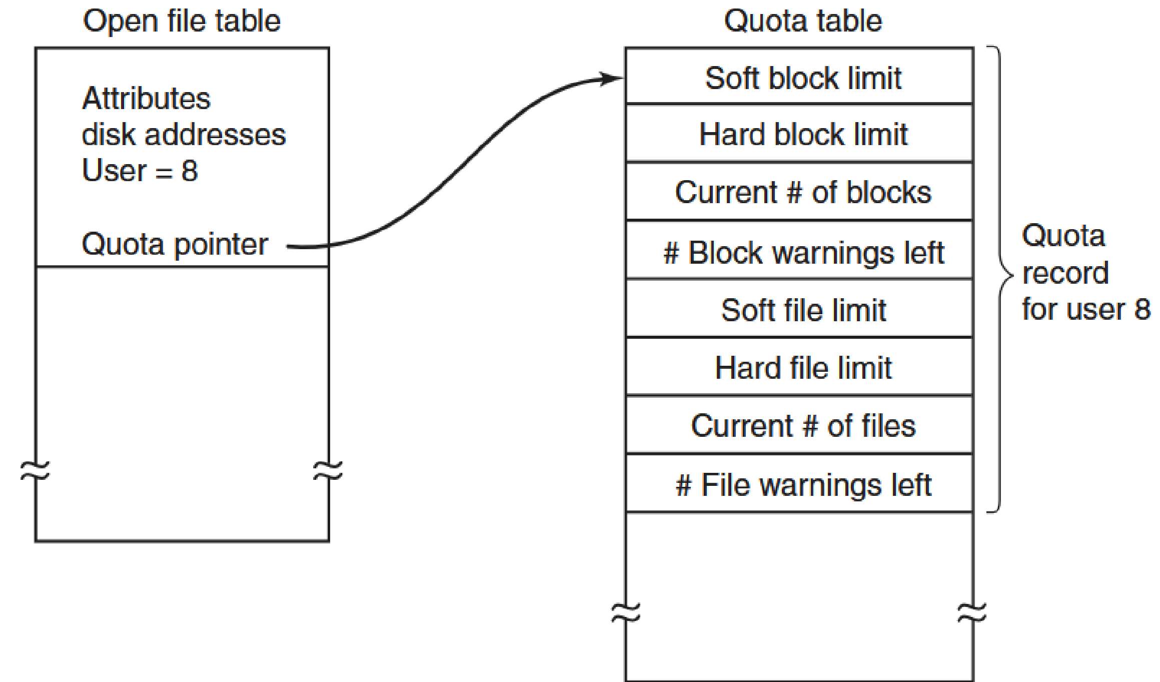


Figure 4-24. Quotas are kept track of on a per-user basis in a quota table.

Use the **du** command to check your disk usage

4. Backups

Problem: Losing data is a larger problem than hardware failures

Solution: Backups provide a way to recover from both disaster and stupidity

Challenges:

- Copying files is time consuming
- Need to make it easy to recover from common mistakes
- Need to make it possible to restore to different points in the past

Idea 1: Recycle bin

Problem: Everyone occasionally deletes files by mistake

Solution: Delay deletion of the file. Instead, mark it as unneeded and put it in a place where the user can restore it later if necessary

Makes it easy for people to restore files without reverting to a full backup

Idea 2: Incremental dumps

Idea: Avoid backing up unchanged files

Solution: Frequently save modified files. Less frequently do a full backup, e.g.

Do Full Dump
- Modify file A
- Remove file B
- Add file C

Do Full Dump
- Modify file C
- Remove A

Why do periodic full dumps?

Improving backup performance

Idea 3: Compression

Make copying data faster by making data smaller

Potential drawbacks:

- Errors are more consequential. A modified bit could corrupt backup.
- Compression algorithm adds extra computation to the backup

Idea 4: Snapshots

Quickly backup critical data structures (like inodes)

Later, copy the associated data files

If a file is edited in the meantime, store a backup of the original

Implementing backup systems

A **physical dump** starts at block 0 and copies each block in sequence

- Simple to save files and to restore them
- Problems
  - saves unused blocks, unchanged files
  - might save “bad blocks” or blocks that should be ignored, such as system files
  - Difficult/slow to restore a single file on request

A **logical dump** starts at a base directory and recursively dumps all files/directories that have changed since the last backup.

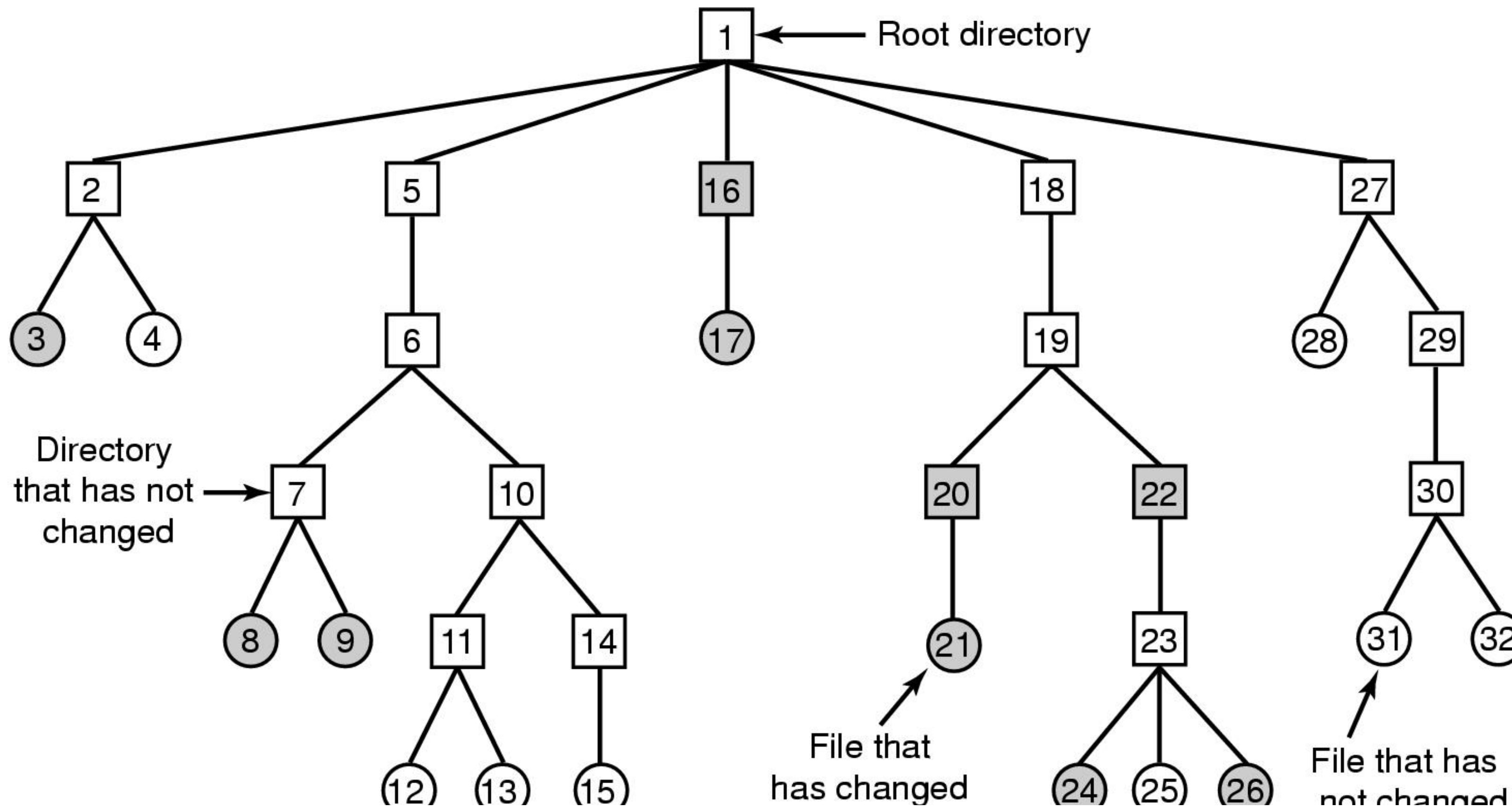
Logical dump algorithm

Initialize a data structure with a bitfield entry for each file/directory

Phases

1. Mark all modified files. Mark all directories that contain a modified file. (Why?)
2. Scan marked inodes and dump directories, along with their owner and dates. Update bitfield to indicate these are finished.
3. Scan marked inodes and dump files, along with their owner and date.

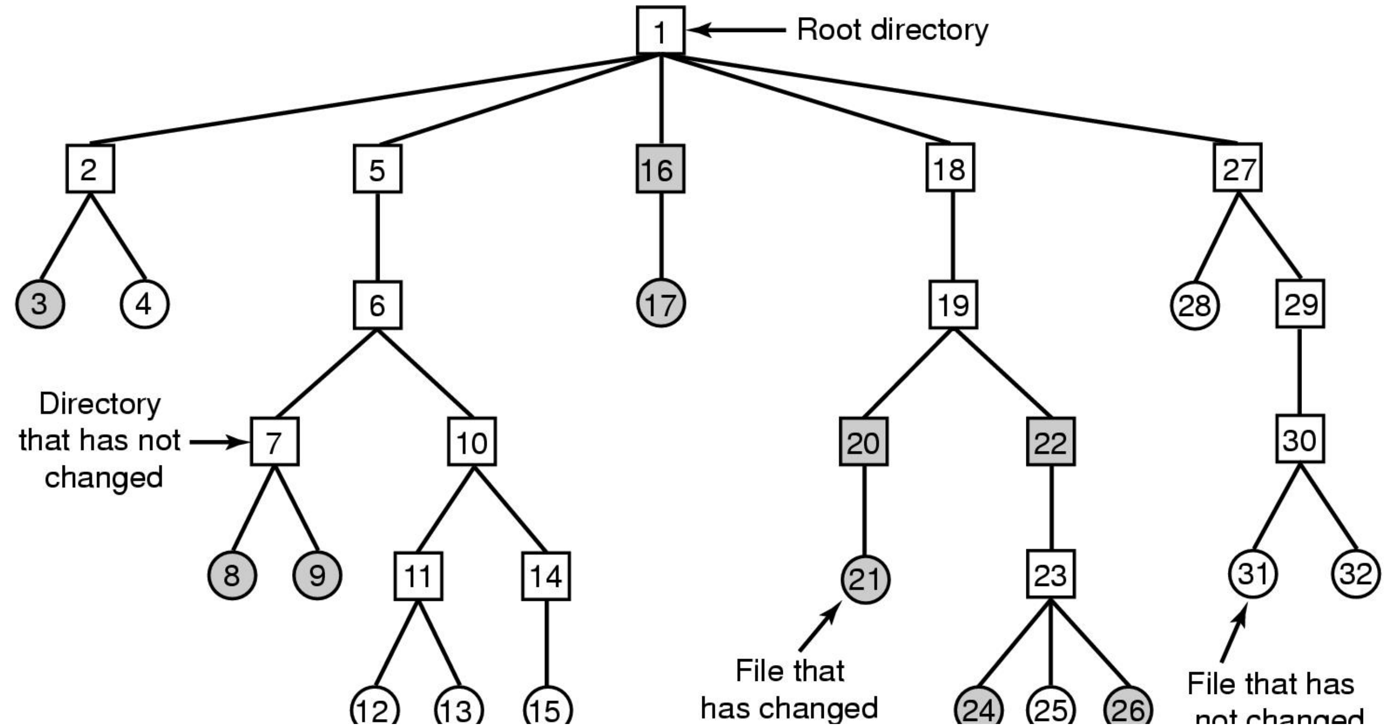
Example: Logical dump



- A file system to be dumped
  - squares are directories, circles are files
  - shaded items, modified since last dump
  - each directory & file labeled by i-node number

Example: Logical dump

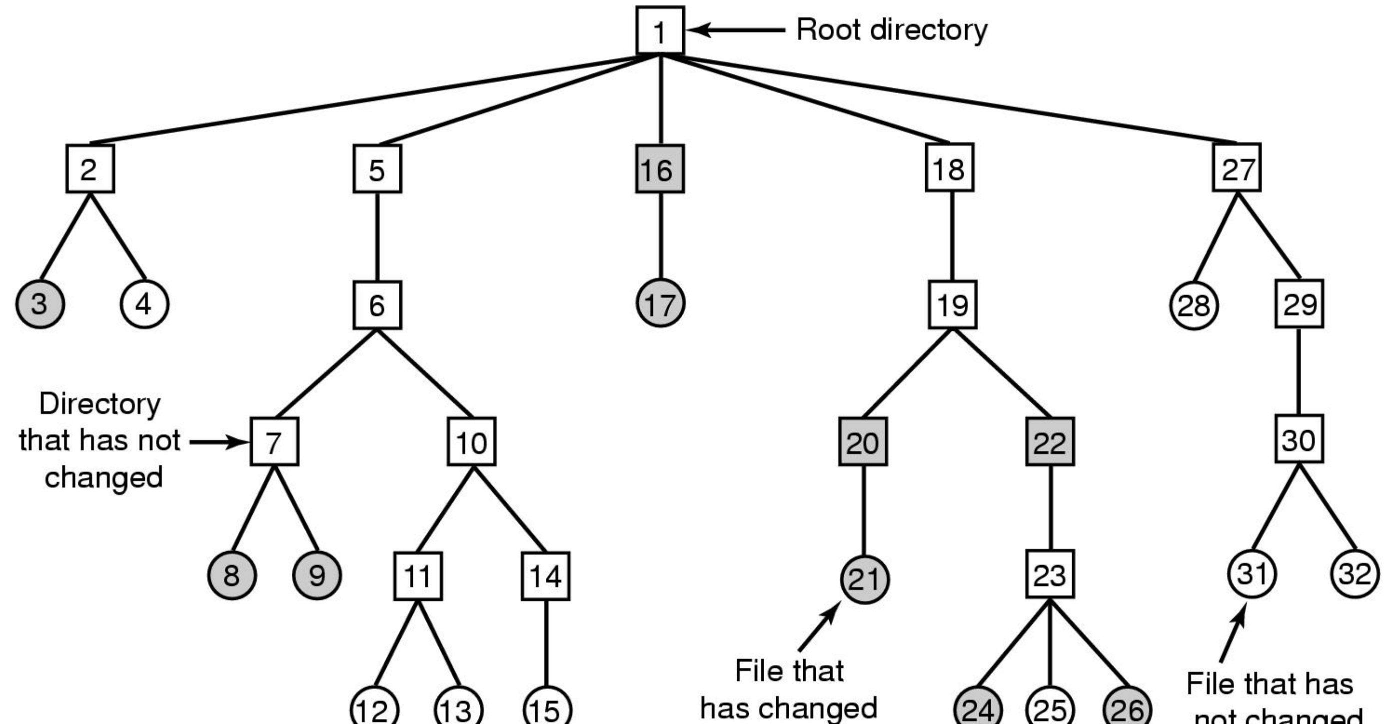
Suppose the following array represents a bitmap of flags that indicate whether a file/directory should be marked for backup (phase 1).

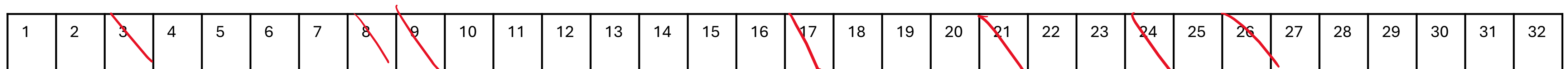


1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

Example: Logical dump

Show how the bitmap of fields would look like after phase 2 of the algorithm, e.g. after saving the directories





1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

Notes: Logical dumps

Directories containing modified files must be saved, even if they haven’t been modified themselves.

Be careful not to duplicate links

Be careful to skip OS/System files

Restoring a logical dump

Phases

1. Create an empty file system
2. Restore directories first (these will be listed first on the dump disk)
3. Restore files

5. Consistency checking

Problem: How can we recovery when the file system becomes in an inconsistent state?

For example, perhaps due to a crash, hardware damage, etc

Idea: Check blocks and files for inconsistent state data

Example: if the number of links in an inode does not match the contents of the directories on the system, something is wrong

Example: if two files refer to the same block, something is wrong

What else could be wrong?

Consistency checking with inodes: Blocks

Keep track of two tables

- Table 1: References to blocks from files
- Table 2: References to blocks from the free list

For each inode

- for each block in inode
  - increment its count by 1 in table 1

For each block in the free list

- Increment its count by 1 in table 2

Example: Consistency checking blocks

Suppose our two tables look as follows

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
Free blocks	0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1

(a)

Does this look consistent?

Example: Consistency checking blocks

Suppose our two tables look as follows

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
Free blocks	0	0	0	0	1	0	0	0	0	1	1	0	0	0	1	1

(b)

Does this look consistent?

Example: Consistency checking blocks

Suppose our two tables look as follows

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
Free blocks	0	0	1	0	2	0	0	0	0	1	1	0	0	0	1	1

Does this look consistent?

Example: Consistency checking blocks

Suppose our two tables look as follows

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Blocks in use	1	1	0	1	0	2	1	1	1	0	0	1	1	1	0	0
Free blocks	0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1

Does this look consistent?

Consistency checking with inodes: Directories

Create a table containing a count for each file. If a directory references this file, increment its count.

After scan, compare the number of references to the number of links in each file

- If links > count, set links = count
- If count > links, set links = count

What would happen if we didn't fix these types of errors?

6. Performance

Problem: Access to permanent storage is much slower than accessing memory

Solutions:

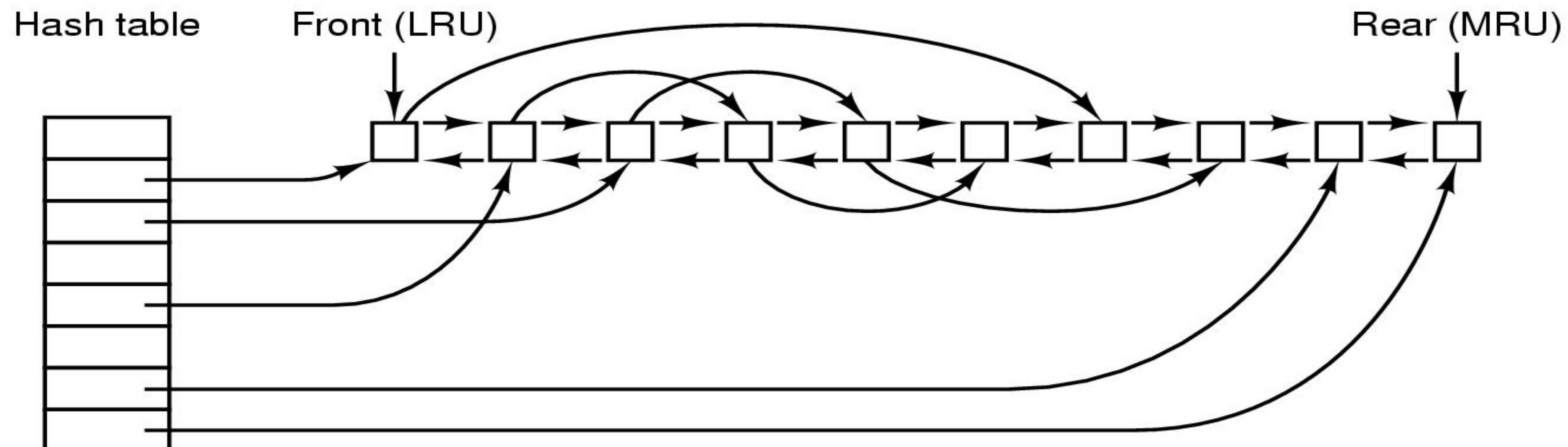
- Caching
- Block read-ahead
- Reducing Disk-Arm Motion

Performance: Caching

Idea: Store frequently accessed filedata in memory

Basic data structure:

- Maintain a doubly linked list of blocks
- Use hashing so that a block can be quickly accessed from its device/address
- LRU implementation possible: every time a block is accessed move it to the back of the list



Performance: Caching

For file system LRU need to consider two things

- Is the block going to be needed again soon?
- Is the block essential to the consistency of the file system?

Approach: Consider the data in the block (e.g. inode, indirect block, full data block, partially full data block, directory block, etc)

- Update LRU queue based on type
    - Example: Partially full data blocks are likely to be modified so put them at the end of the list
- To maintain consistency, write modified data to disk asap
    - Non-data blocks are essential for consistency -> write modifications to disk immediately
    - Data blocks should also be written soon after modification so user doesn’t lose data
    - Caches with this feature is called a **write-through cache**

Performance: Block read-ahead

Idea: When fetching block k, also fetch block (k+1)

Do a sneaky check for block k+1 when block k is accessed. If it's not there, request it.

Files can be marked as sequential or random access files based on their usage patterns (e.g. whether seek is used)

Performance: Reducing disk arm motion

Idea: Put data that is likely to be accessed together close together on disk

Example: Place inodes on the same cylinder as their data

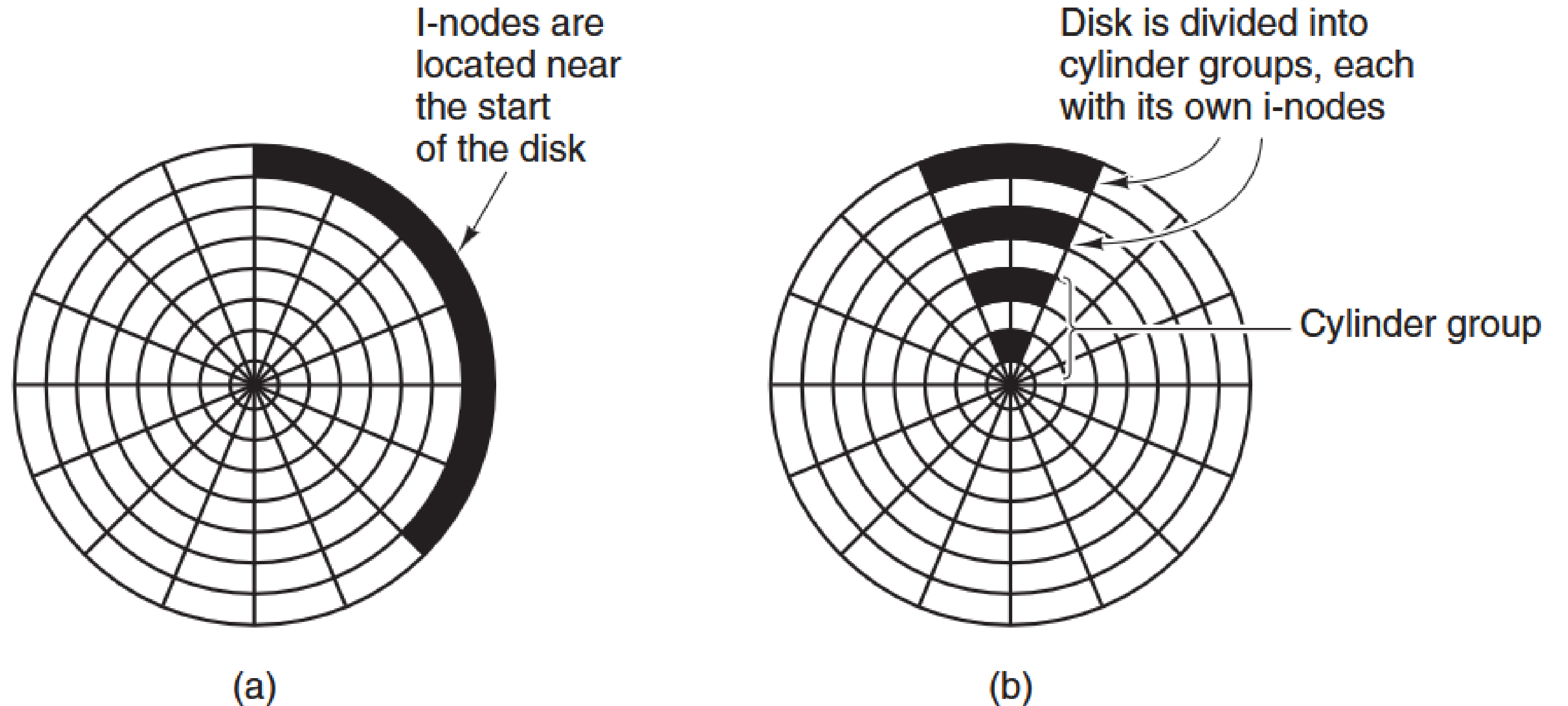


Figure 4-29. (a) I-nodes placed at the start of the disk. (b) Disk divided into cylinder groups, each with its own blocks and i-nodes.

7. Defragmentation

Problem: Disks preform best when data access is sequential

Defragmenting a disk rearranges the block of a file so they are sequential

Works best when there is a lot of free space that can be used for storing blocks while others are moved.

As with backups, some OS files should not be moved/touched

7. Defragmentation: Not for SSD!

Solid-state drives (SSD) performance does not rely on sequential data, e.g. random access is just as fast as sequential access

SSD drives have a limited number of writes before they wear out

- SSDs still last a long time because the hardware tries to spread out the wear and tear evenly throughout the drive

Defragmenting an SSD is a waste of time and needlessly puts wear on the drive.