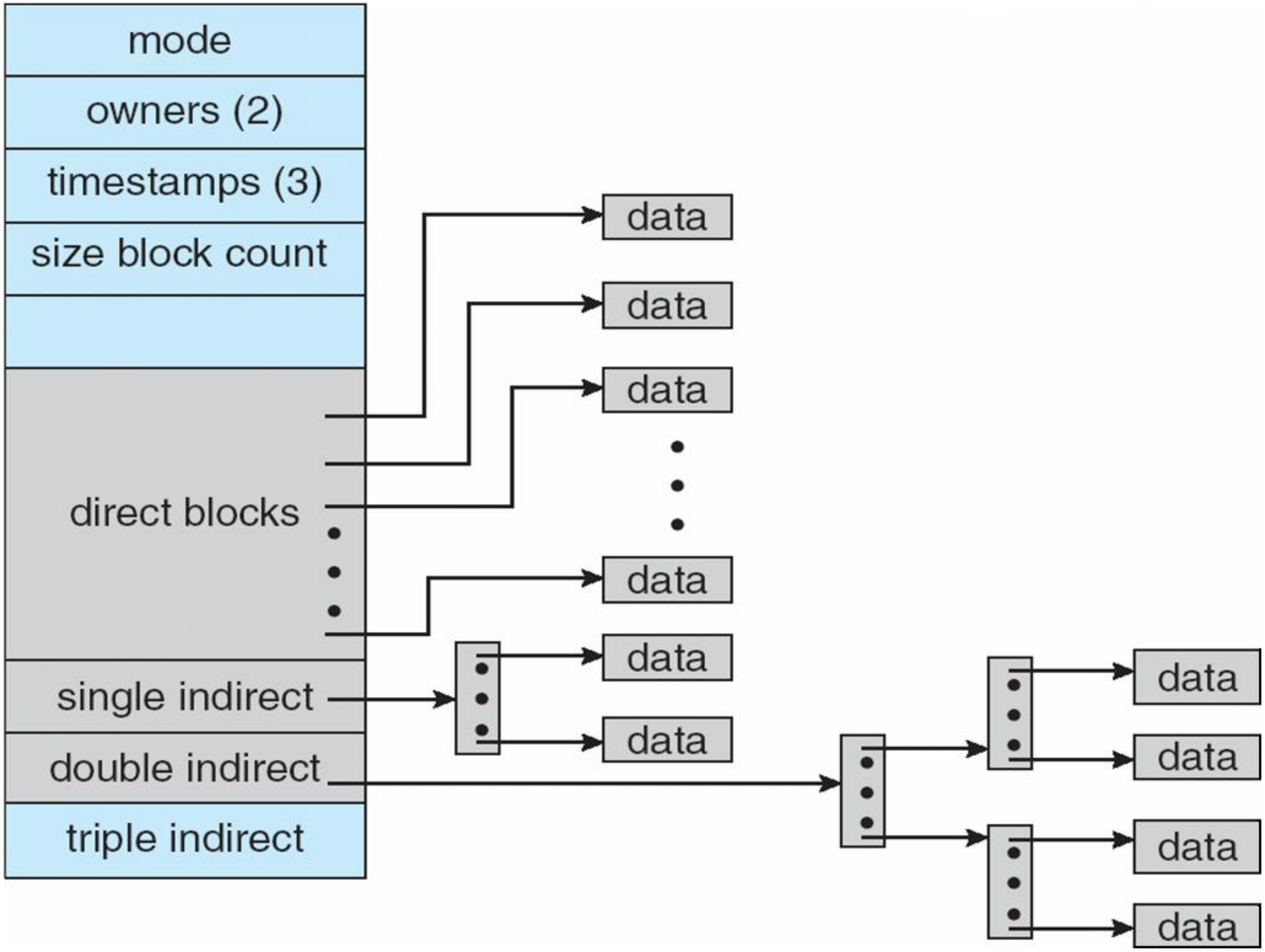
CS416 – Filesystems
Buffer Cache, Page Cache, FFS

CS 416: Operating Systems Design, Spring 2011

Department of Computer Science
Rutgers University

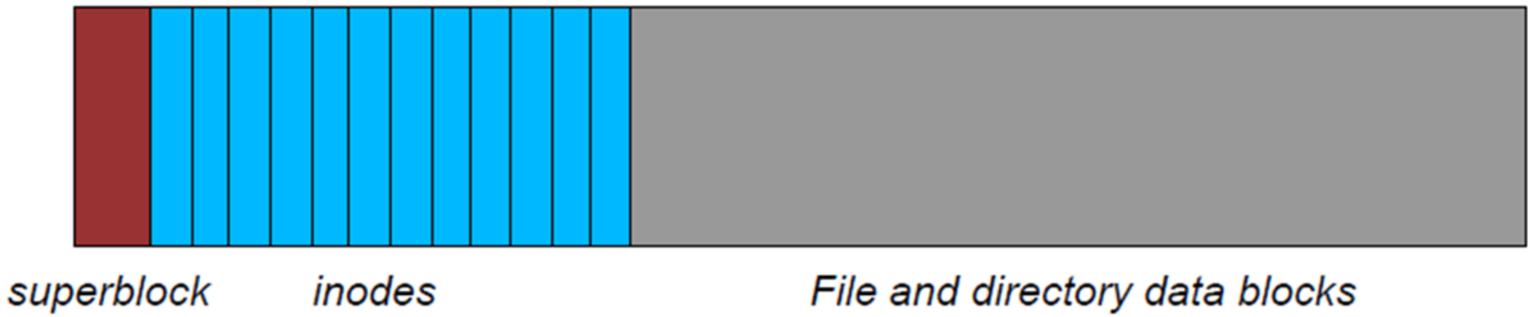
Rutgers Sakai: 01:198:416 Sp11
(https://sakai.rutgers.edu)

inode structure: UNIX Filesystem



Size = 64 Bytes, (10 direct blocks, 1 single indirect, 1 double indirect and 1 triple indirect

Recall Filesystem Layout…

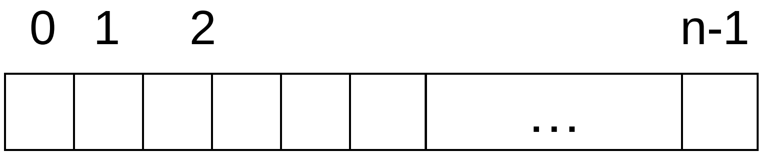


Superblock:

- Filesystem Type
- Number of Free inodes
- Number of Free blocks
- Mount Status of the filesystem
- Block Size
- Location of first inode ("/")
- Pointer to the first free block.
- Array to represent free inodes

Free-Space Management

- Bit vector (*n* blocks) - Held in memory



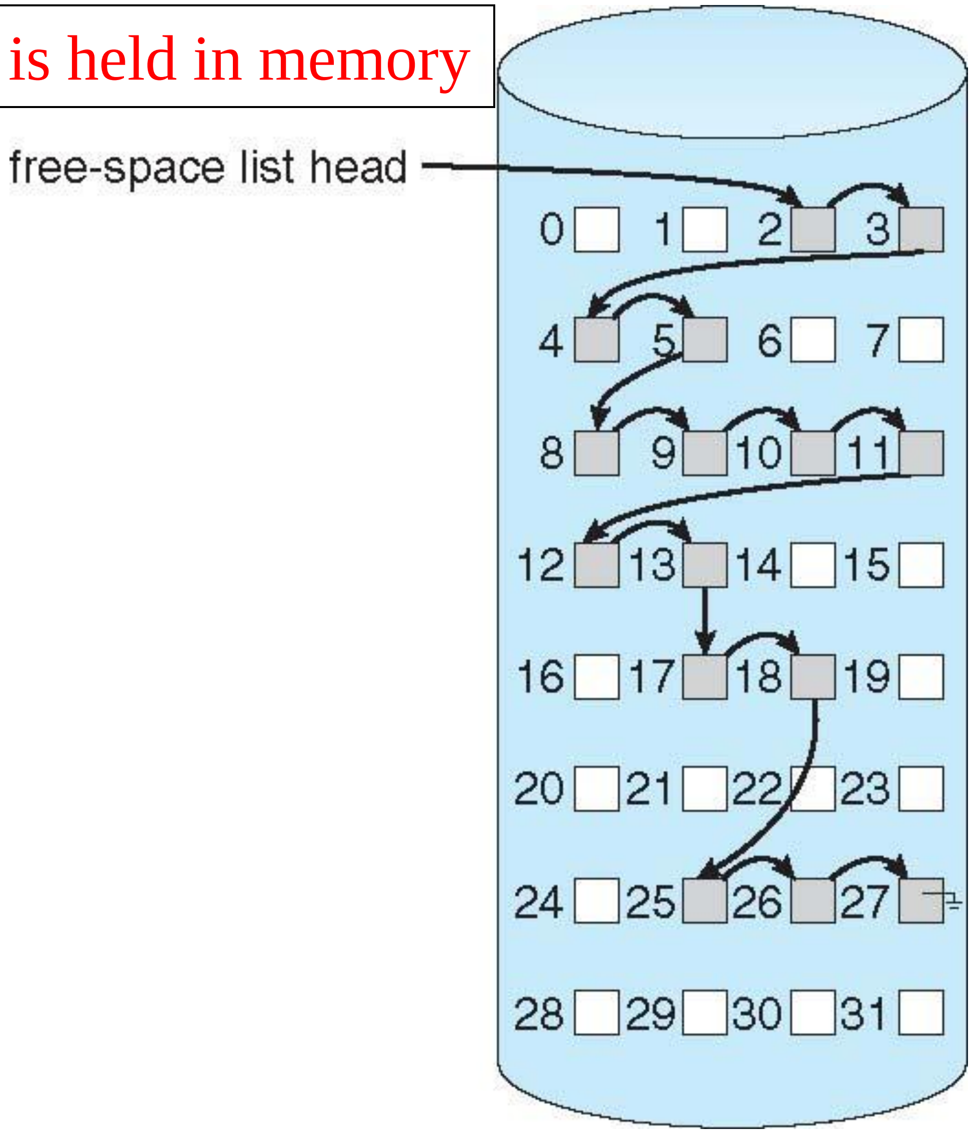
$$\text{bit}[i] = \begin{cases} 0 \Rightarrow \text{block}[i] \text{ free} \\ 1 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

Block number calculation

(number of bits per word) *
(number of 0-value words) +
offset of first 1 bit

Linked Free Space List on Disk

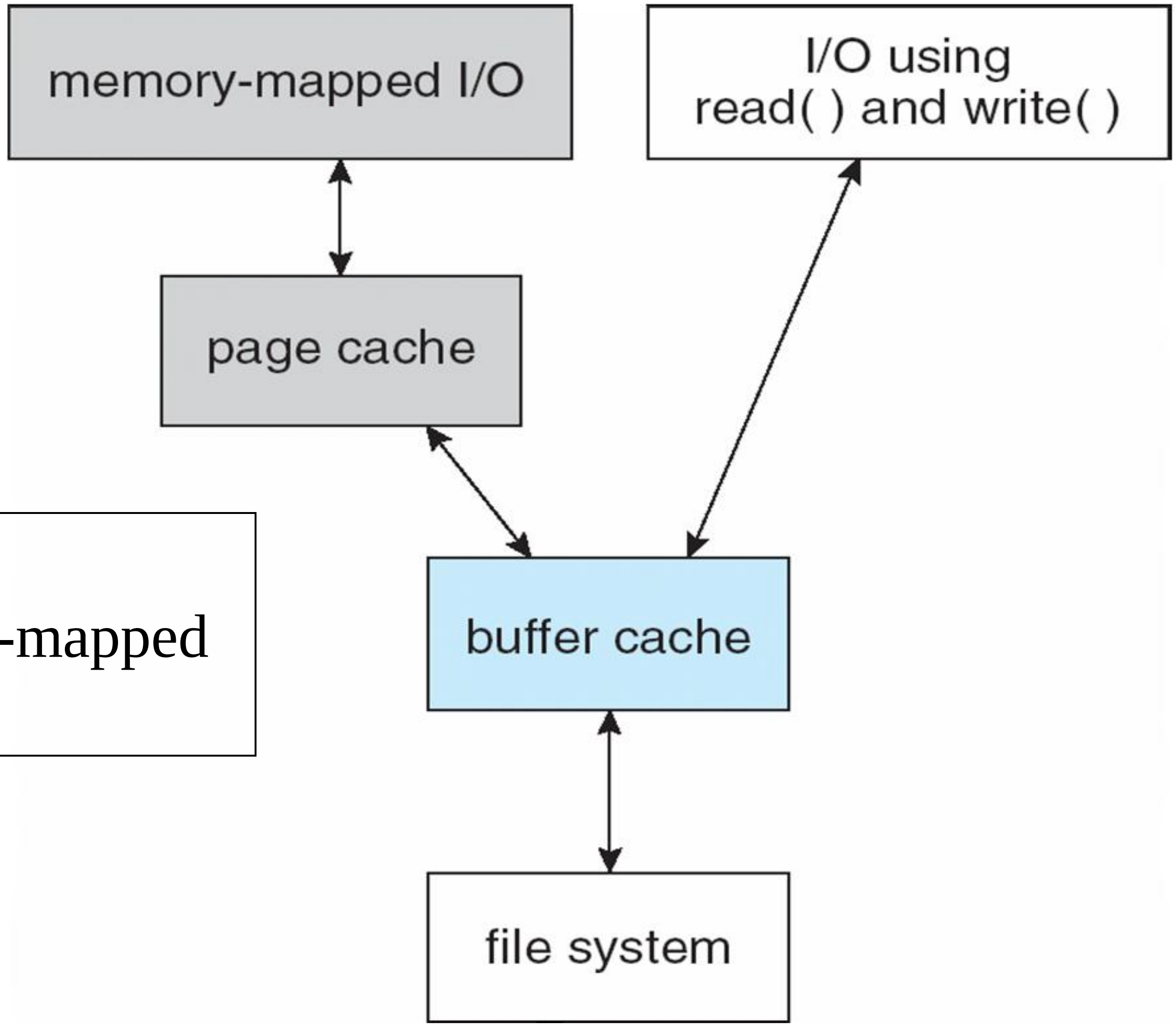
This pointer is held in memory



Page Cache

- A **page cache** caches pages rather than disk blocks using virtual memory techniques
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure

I/O Without a Unified Buffer Cache

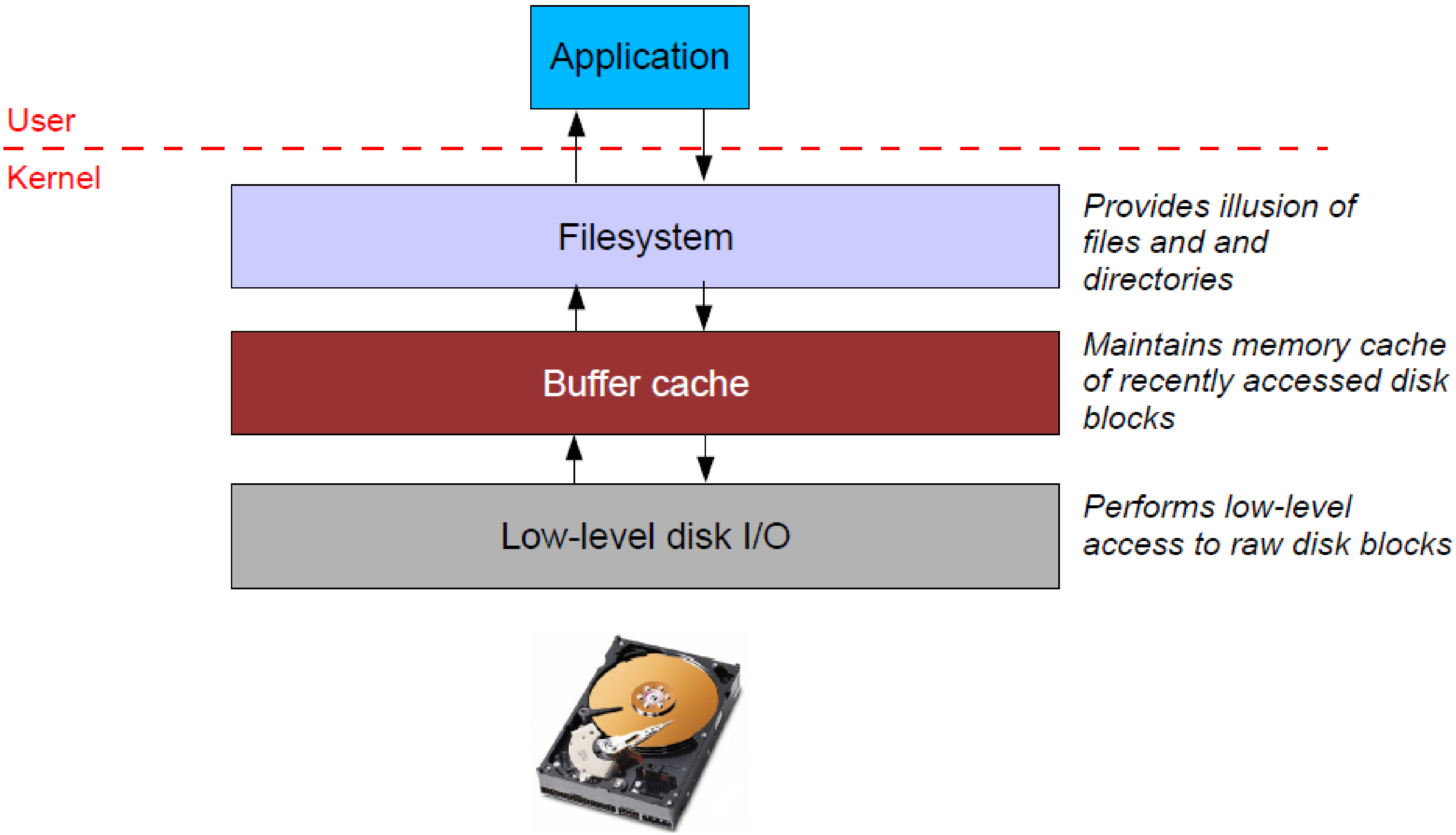


What is Memory-mapped I/O ?

File System Caching

Most filesystems cache significant amounts of disk in memory

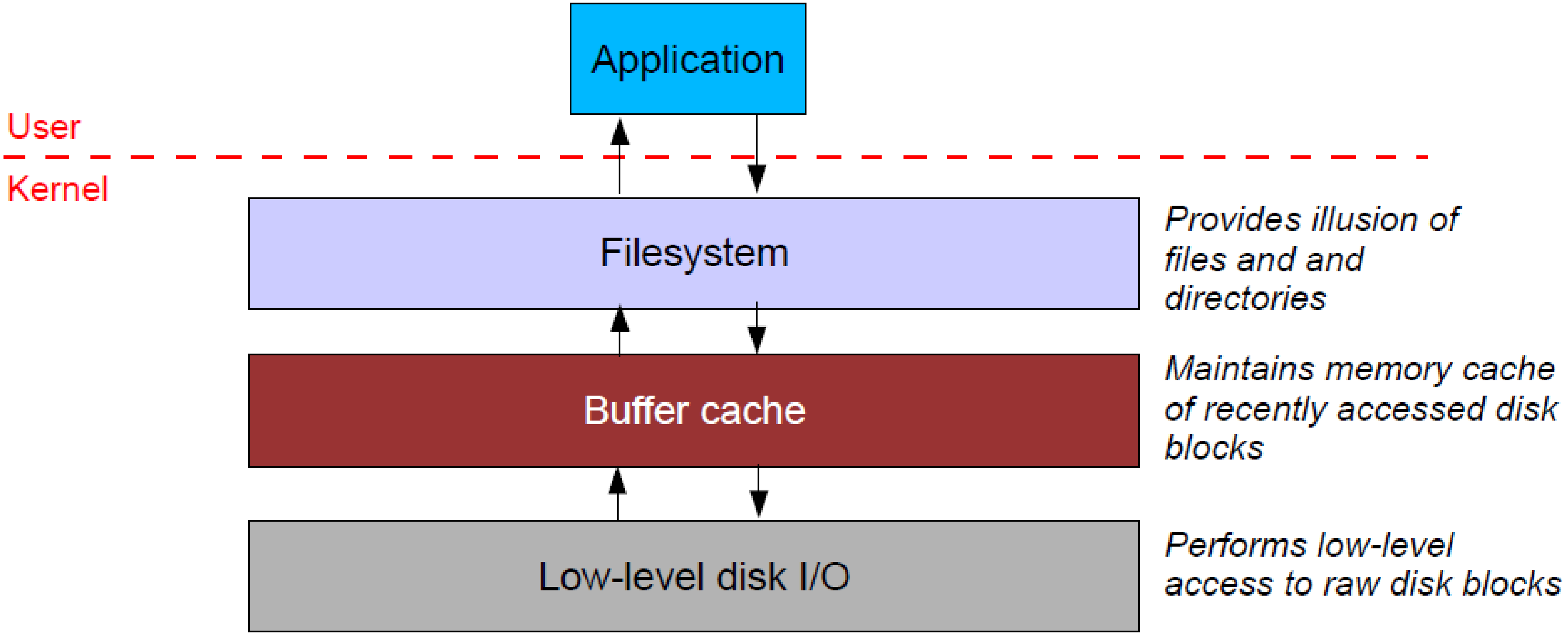
- e.g., Linux tries to use all “free” physical memory as a giant cache
- Avoids huge overhead for going to disk for every I/O



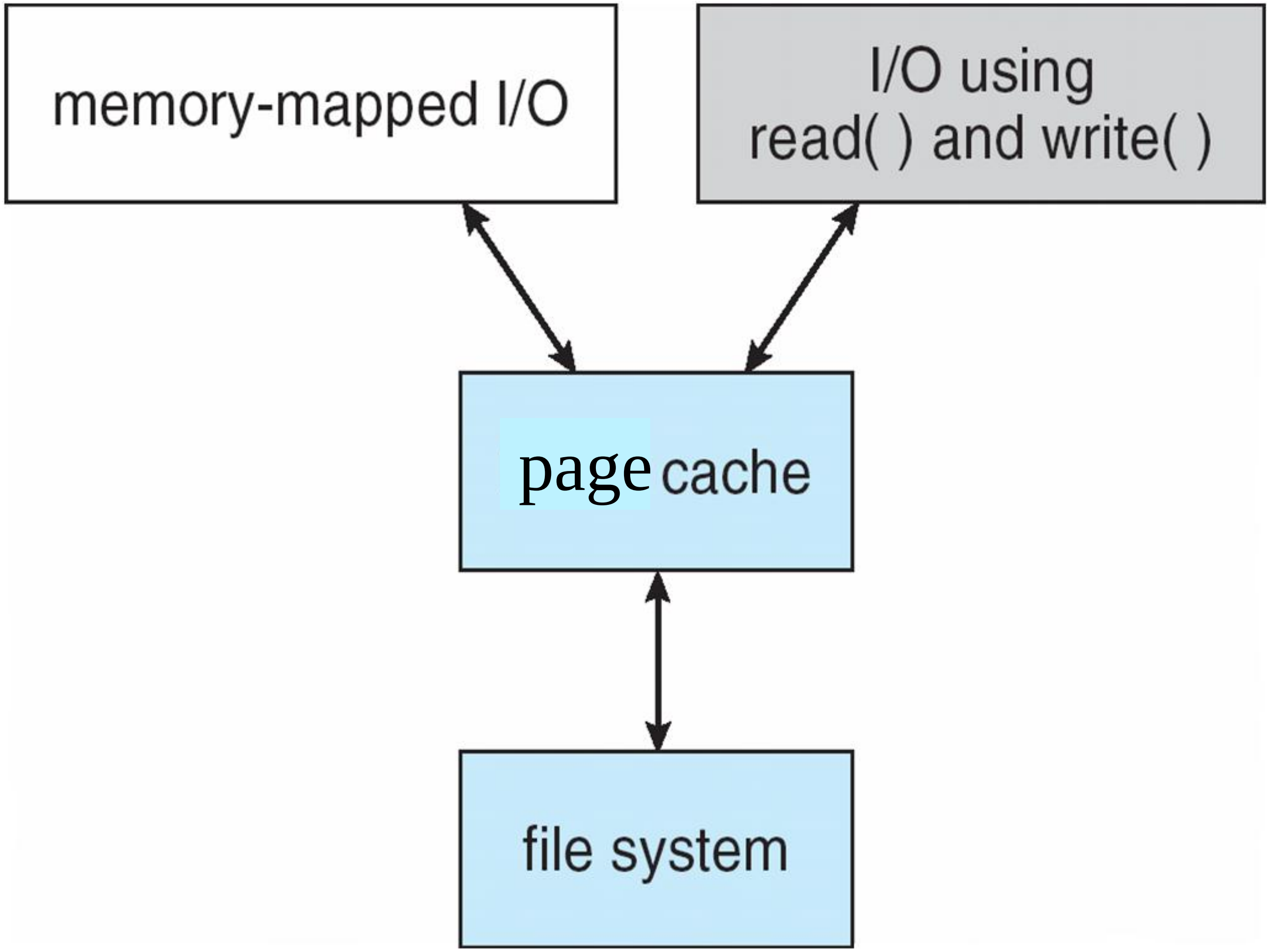
Caching Issue

Reliability issues

- What happens when you write to the cache but the system crashes?
- What if you update some of the blocks on disk but not others?
  - *Example: Update the inode on disk but not the data blocks?*
- **Write-through cache:** All writes immediately sent to disk
- **Write-back cache:** Cache writes stored in memory until evicted (then written to disk)
  - *Which is better for performance? For reliability?*



I/O Using a Unified Buffer Cache



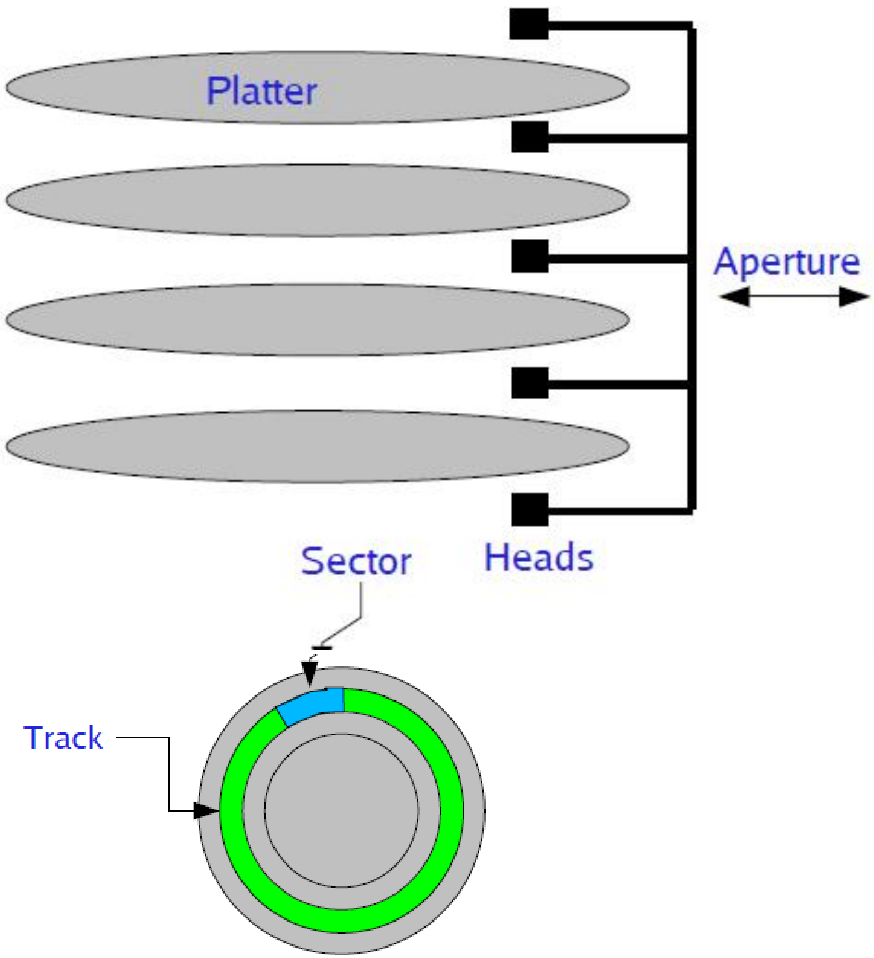
Berkley Fast File system (FFS)

- Motivated by performance problems with older UNIX filesystems:
  - Older UNIX FS had small blocks (512 bytes)
  - Free list was unordered; no notion of allocating chunks of space at a time
  - inodes and data blocks may be located far from each other (long seek time)
  - Related files (in same directory) might be very far apart
  - No symbolic links, limited filenames (12 chars), no quotas
- Main goal of FFS was to improve performance:
  - Use a larger block size – *why does this help??*
  - Allocate blocks of a file (and files in same directory) near each other on the disk
- Entire filesystem described by a *superblock*
  - Contains free block bitmap, location of root directory inode, etc.
  - Copies of superblock stored at multiple locations on disk (for safety)

Disk Layout

Disks consist of one or more *platters* divided into *tracks*

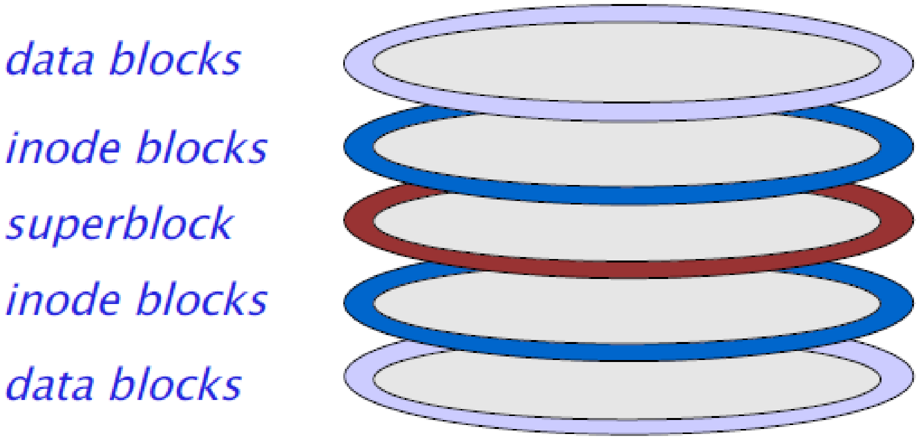
- Each platter may have one or two *heads* that perform read/write operations
- Each track consists of multiple *sectors*
- The set of sectors across all platters is a *cylinder*



FFS Cylinder Groups

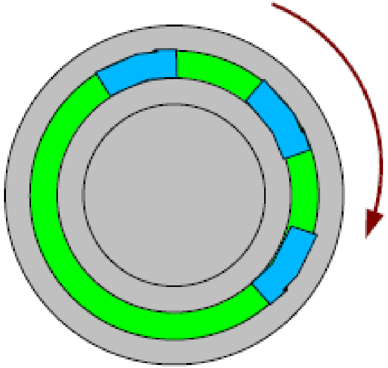
Store related blocks on nearby tracks but on different platters

- That is, a whole *group of cylinders:*



Allocate blocks in a rotationally optimal fashion:

- Try to estimate rotation speed of disk and allocate next block where the disk head will happen to be when the next read will be ready!



Colocating inodes and directories

Problem: Reading small files is slow. Why?

- What happens when you try to read all files in a directory (e.g., "ls -l" or "grep foo *") ?
- Must first read directory.
- Then read inode for each file.
- Then read data pointed to by inode.

Solution: Embed the inodes in the directory itself!

- Recall: Directory just a set of <name, inode #> values
- Why not stuff inode contents in the directory file itself?
    - *What filesystem feature do we possibly give up when doing this?*

Problem #2: Must still seek to read *contents* of each file in the directory.

- Solution: Pack all files in a directory in a contiguous set of blocks.

Colocating inodes and datablocks

- Each Cylinder Group has
  - A bitmap of free inodes
  - A bitmap of free datablocks
  - Inodes and data blocks are allocated from the same cylinder group to reduce disk seek time

FFS Block Size

Older UNIX filesystems used small blocks (512B or 1KB)

- Low disk bandwidth utilization
- Maximum file size is limited (how many blocks each inode could keep track of)

FFS introduced larger block sizes (4KB)

- Allows multiple sectors to be read/written at once
- Introduces *internal fragmentation*: a whole block may not be used

Fix: Block “fragments” (1KB)

- The last block in a file may consist of 1, 2, or 3 fragments
- Fragments from different files stored on the same block
  - *inode needed to store both block ID and “fragment index” of fragment within block*

Longer File Names

- Directory Structure changed to accommodate long names

Unix Directory Entry

<12 bytes Filename> <inode>

Directory Entry 0		
0	4	inode number: 783362
4	2	record length: 12
6	1	name length: 1
7	1	file type: EXT2_FT_DIR=2
8	1	name: .
9	3	padding
Directory Entry 1		
12	4	inode number: 1109761
16	2	record length: 12
18	1	name length: 2
19	1	file type: EXT2_FT_DIR=2
20	2	name: ..
22	2	padding
Directory Entry 2		
24	4	inode number: 783364
28	2	record length: 24
30	1	name length: 13
31	1	file type: EXT2_FT_REG_FILE
32	13	name: .bash_profile
45	3	padding

Symbolic Link

- Unix System only supported Hard Links
  - ln /tmp/foo /usr/foo (The directory for foo in /tmp and /usr point to same inode
  - Therefore cannot span across filesystems
  - Hard links just point to the "inode"
- FFS introduced Symbolic Links
  - Symbolic Links are "files" that store Unix path name to the file (instead of inode number)
  - This path is just added to the current program's search path
  - Therefore, could span across filesystems

Quotas

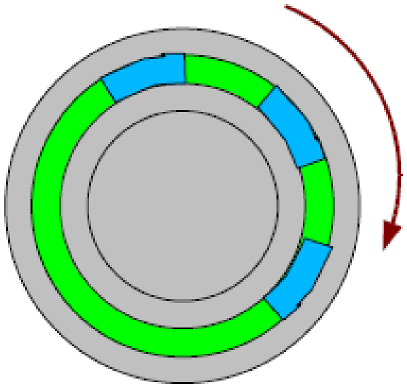
- No Concept of Quotas Existed in the Original Unix

- FFS
  - Soft limit
  - Hard Limit

These are limits on the number of inodes that a given user can use up.

Further Enhancements - Extent Based Transfer

Recall: FFS adds a gap between sectors on a track



- Try to take advantage of rotational latency for performing next read or write operation
- Problem: Hurts performance for multi-sector I/O!
  - *FFS cannot achieve the full transfer rate of the disk for large, contiguous reads or writes.*

Possible fix: Just get rid of the gap between sectors

- Problem: "Dropped rotation" between consecutive reads or writes: have to wait for next sector to come around under the heads.

Hybrid approach - "extents" [McVoy, USENIX'91]

- Group blocks into "extents" or clusters of contiguous blocks
- Try to do all I/O on extents rather than individual blocks
- To avoid wasting I/O bandwidth, only do this when FS detects sequential access
  - *Kind of like just increasing the block size...*