File Caching

ECE595
Nov 13

Y. Charlie Hu

[lec22] UFS concurrency semantics

- What happens when two processes try to write to the same file?
  - What is the programmer's intent?
- What needs to be made atomic in FS impl?
  - AllocateBlock(); FreeBlock()
  - Write() – AllocateBlock() and update inode
- What are naturally atomic?
  - WriteRawData(); ReadRawData()
- Analogy?

Disk Allocation revisited – many low level details

- Finally, UFS design focused on inode
- How to keep blocks for a file together?
- How about inode and data blocks for a file?
  - It is a good idea to keep them close?
  - If so, how?
- How about files in the same directory?
  - e.g. make

True or False

- On Unix, a user process can read/write a dir just like reading/writing an ordinary file, assuming the user has the read/write permission

Roadmap

- Functionality (API)
  - Basic functionality
    - Disk layout
    - File operations (open, read, write, close)
  - Directories
- Performance
  - Disk allocation
  - Buffer cache
  - File system interface
  - Disk scheduling
- Reliability
  - FS level
  - Disk level: RAID

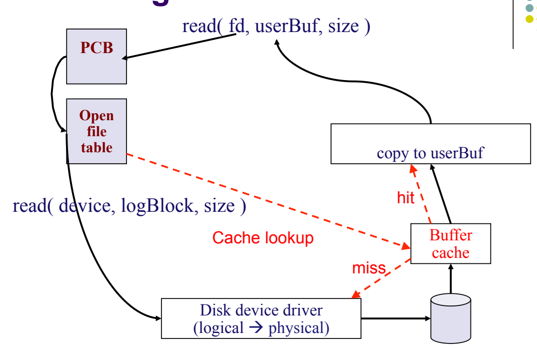
"Principle of locality" once more

- Locality of reference in file accesses
  - Yet another manifestation of the principle of locality
  - What were the earlier instances in this class?
- Keep a number of disk blocks in "the much faster" memory
  - when accessing disk, check the cache first!
- File system buffer caches are maintained in software

How many disk blocks to cache?

- Fixed portion of main memory (BSD)
- Variable portion of main memory (modern Unix) -- processes and file system compete for physical memory
- Pros/cons?

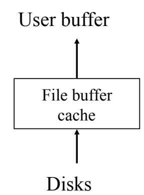
Reading A Block



Modern disk drives are addressed as large one-dimensional arrays of logical blocks

Read operations in presence of buffer cache

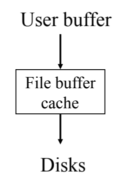
- read(fd, buf, n)
  - On a hit
    - copy from the buffer cache to a user buffer
  - On a miss
    - replacement if necessary
    - read a file block into the buffer cache



- Concurrent reads?

Write operations: Maintaining Consistency

- write(fd, buffer, n)
  - On a hit
    - write to buffer cache
  - On a miss?



- Concurrent writes?
- Concurrent read/write?

File persistence under file caching

- Problem: fast cache memory is volatile, but users expect disk files to be persistent
  - In the event of a system crash, dirty blocks in the buffer cache are lost !
- Example 1: creating "/dir/a"
  - Allocate inode (from free inode list) for "a"
  - Update parent dir content – add ("a", inode#) to "dir" data block
- If crash happens bf neither is flushed to disk?

File persistence under file caching

- Problem: fast cache memory is volatile, but users expect disk files to be persistent
  - In the event of a system crash, dirty blocks in the buffer cache are lost !
- Example 2: append a block to a file
  - Allocate data block (from free block list)
  - Update inode content (in memory copy)
  - Write to new data block (in buffer cache)
- If crash happened before ?

File persistence under file caching

- Solution 1: use write-through cache
  - Modifications are written to disk immediately
    - (minimize “window of opportunities”)
  - No performance advantage for disk writes
- Example 2: append a block to a file
  - Allocate data block (from free block list on disk)
  - Update inode content (to disk copy)
  - Write to new data block (to disk)

File persistence under file caching

- Solution 2: limit potential data loss (Unix)
  - Write-through caching for metadata (inodes, directories, free block list)
  - Write back
    - dirty data blocks after no more than 30 seconds
    - all dirty blocks during file close
  - Worse case damage?
- Example 2: append a block to a file
  - Allocate data block (from free block list on disk)
  - Update inode content (to disk)
  - Write to new data block (in buffer cache)

File caching implementation

- Two major issues
  - Buffer cache replacement
    - What problem does this resemble?
    - How are they different?
    - Implications?
  - Competition with VM for main memory
    - Static partitioning during kernel configuration (BSD)
    - Dynamic adjustment of partitioning during runtime,
      - e.g. keep miss frequencies of VM and buffer cache, and try to balance them (e.g. Linux)

Buffer cache replacement

- A classic OS research topic
- New ideas still come out
- Recency / frequency based
  - (exact) LRU, MRU, LRU-K, FBR, LRFU, etc.
- Pattern based - manual
  - User inserted / compiler generated
  - Ex: What is optimal for sequential access?
- Pattern based - automatic
  - DEAR: per appl pattern classification
  - UBM: per file pattern classification
  - PCC: per call-site classification [Gniady/Butt/Hu OSDI ‘04]

Other performance optimizations

- Read-ahead (e.g. Linux)
  - For sequential access, read the requested block and the following N blocks together (why is this a good idea?)
- Write-behind:
  - Start disk write, but don’t make application wait until the disk operation completes
- Allow overlap of a process’s computation with its own disk I/O (e.g. AIO in FreeBSD)