CS 347M (Operating Systems Minor)

Spring 2022

Lecture 19: Filesystem Implementation

Mythili Vutukuru

CSE, IIT Bombay

Disk buffer cache

- File data that is read from hard disk is retained in memory for some time in the disk buffer cache = memory pages that cache recently read disk data
- Any changes to disk data is made in the cached copy of disk buffer cache first, then written to disk later
  - Write-through cache: changes written to disk immediately (synchronous writes)
  - Write-back cache: changes written to disk after some delay (asynchronous writes)
  - Write-back cache has better performance, but can lose data in case of power failure
- Benefits of disk buffer cache
  - Improved performance due to fewer disk accesses
  - Merge changes when multiple processes modify same file data
- Most OS allocate unused physical memory to disk buffer cache
  - Some applications doing their own optimizations can bypass cache

Read system call

```
fd = open("/home/foo/a.txt")
char buf[64]
n = read(fd, buf, 64)
```

- Input is file descriptor, user memory to read into, number of bytes to read
  - Use file descriptor array index, access open file table entry, then inode
  - Based on offset, identify which data block(s) of file to read using inode information
  - Check if file data block(s) present in disk buffer cache
    - If cache miss, device driver issues read command to hard disk, process is moved to blocked state, OS will context switch to another process
    - When read completes, device controller will DMA the block(s) into an empty buffer in disk buffer cache, raises interrupt
    - OS handles interrupt, marks process as ready to run, scheduler will switch to process in future
  - Copy requested number of bytes from data block(s) in disk buffer cache into user-provided memory buffer
  - User code resumes, system call returns number of bytes actually read, or error
    - Actual bytes may be less than requested, e.g., end of file

Write system call

- Input is file descriptor, user memory buffer containing data to be written, number of bytes to write
  - Using file descriptor and inode, identify which data block(s) of file to write into
  - If we are writing beyond end of file, file size expands, new blocks needed
    - Allocate new data blocks for file on disk (update free list or bitmap)
    - Add new data block numbers into file inode
  - Locate data block(s) present in disk buffer cache
    - If not, read data block(s) into buffer cache first
  - Copy requested number of bytes from user memory buffer into data block(s) cached in disk buffer cache, cached block is now marked "dirty"
    - Write-through cache: synchronously write to disk immediately
    - Write-back cache: asynchronously update disk copy later
  - User code resumes (after delay in case of sync write, immediately for async write), system call returns number of bytes actually written, or error

Linking and unlinking

- Same file can be "linked" from different directories with different filenames using link system call
  - When file created, it is linked to its parent directory for first time
  - Subsequently, we can link to same file data from another directory also
- Hard linking: add entry in new directory, mapping new filename to old inode
  - If file deleted from old pathname, can still access it from new pathname
  - Link count of file in inode captures the number of such "links" to file inode
- Soft linking: add entry in new directory, mapping new filename to old filename
  - If file deleted from old pathname, soft link is "broken"
- Unlink system call: remove directory entry of a file from a particular directory
  - If this is last "link" to the file, the file is deleted from disk

Crash consistency

- Every system call updates multiple disk blocks
  - Example: when we append data to a file, we change data block, inode block, bitmap, ..
- All changes to disk blocks are first made in memory (disk buffer cache), then written to disk (synchronously or asynchronously)
  - Even metadata blocks (inode) are updated first in disk buffer cache
- If power failure happens in the middle of a system call, memory changes will be lost, disk can be only partially updated, may cause inconsistency in file data
  - Example: new data block written to disk, but not added to inode (written data is lost)
  - Example: new data block number added to inode, but data block contents not written (file contains garbage data)
- Crash consistency: how to ensure filesystem is consistent after a power failure?
  - Problem exists even with write-through disk buffer cache, but more prominent with write-back cache

Filesystem checkers

- Programming tip for crash consistency: always update data blocks on disk first before updating metadata blocks
  - Better to write data block and not link from inode (lost data), rather than link from inode first and fail to update data block (garbage in file)
- Even with above tip, inconsistency can still occur, especially when multiple metadata blocks need to be updated
  - Example: bitmap updated to mark data block as used, inode updated to add pointer to data block, which metadata change to write to disk first?
- File system checking tools (e.g., fsck) check inconsistencies in metadata blocks after reboot and fix the blocks to make them consistent
  - Example: data block marked as used in bitmap, but not present in any inode, so mark as free
- What we want: atomicity (all changes pertaining to a system call happen all at once together or none happens at all)

Logging / journaling

- Logging/journaling: common technique for atomicity in systems
  - Can be applied to guarantee crash consistency in filesystems also
- How to add logging to any filesystem?
  - All changes to be made to disk blocks are first written to a log on disk, original disk blocks are not touched
  - After all changes are logged to disk, special commit entry written to log
  - Next, changes are applied to the original disk blocks, log entries cleared
  - If crash happens before log is committed, then no changes are made to any disk block, it is as if system call never happened
  - If crash happens after log is committed, but before changes applied to original disk blocks, then log is replayed upon reboot and changes are completed