

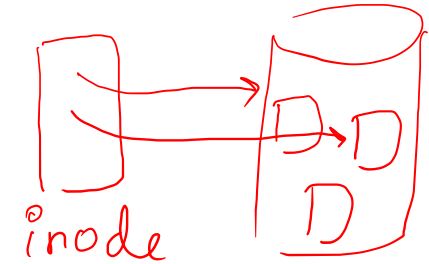
Design and Engineering of Computer Systems

Lecture 17: Filesystem Implementation

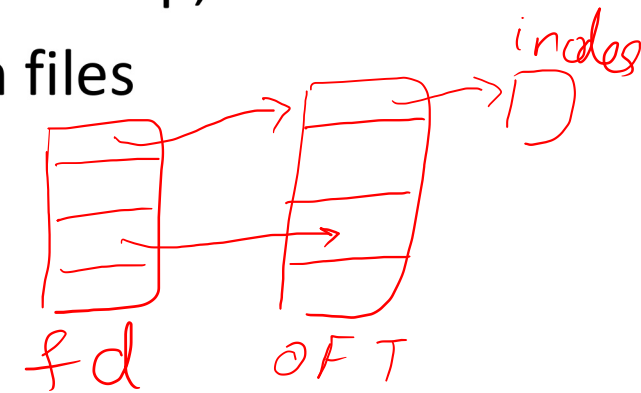
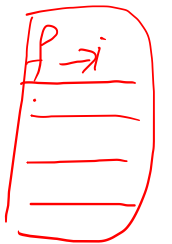
Mythili Vutukuru

IIT Bombay

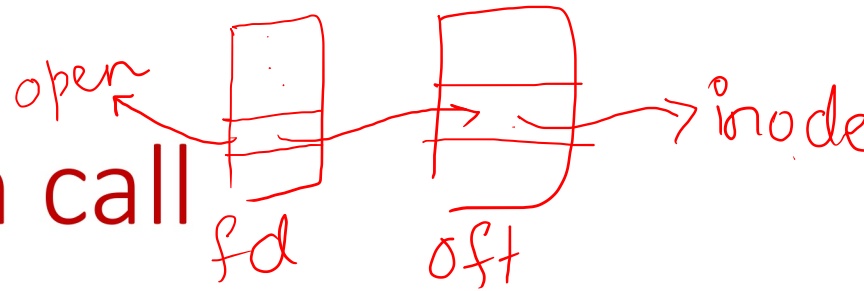
Recap of filesystems



- File data is stored non-contiguously on disk blocks, index node (**inode**) keeps track of all block numbers containing file data
- Inode number of a file uniquely identifies a file in a filesystem
- Directory also a special file with mappings from filename to inode number
 - Recursively traverse directories in pathname to locate inode number of file
- On disk layout of filesystem: superblock, data blocks, inodes, bitmap,...
- OS has many in-memory data structures to keep track of open files
 - In-memory copy of inode (cached from on-disk inode)
 - Open file table (list of files opened in entire system)
 - File descriptor array (files opened by specific process, part of PCB)
 - Disk buffer cache (cache of recently accessed disk blocks)

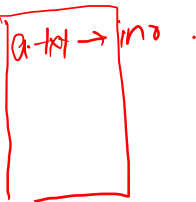


Open system call



```
fd = open("/home/foo/a.txt", flags)
read(fd, ..)
write(fd, ..)
close(fd)
```

- Takes file pathname and other flags as input, returns file descriptor of file
 - Traverse pathname in directory tree, find inode number of file
 - Create a new file if one doesn't exist (depending on flags given to open) → allocate new inode, add mapping from filename to inode number in parent directory
 - Copy inode of file into memory from disk
 - Create new open file table entry, with pointer to in-memory inode
 - Allocate free entry in file descriptor array, store pointer to open file table entry
 - Return index of newly allocated file descriptor array entry
- Every process has 3 files open by default: standard input, output, error (entries 0, 1, 2 in file descriptor array)
 - Subsequent open files will get next free entries in file descriptor array
- Close system call deletes file descriptor and open file table entries

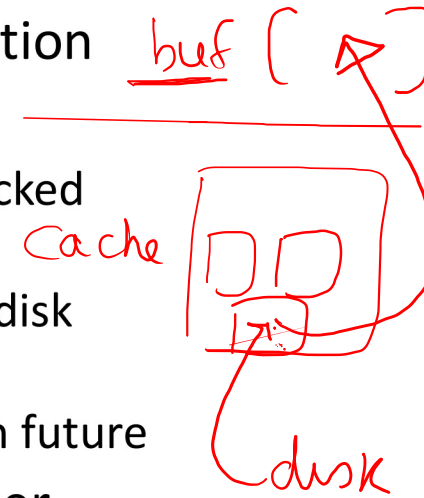


Read system call

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

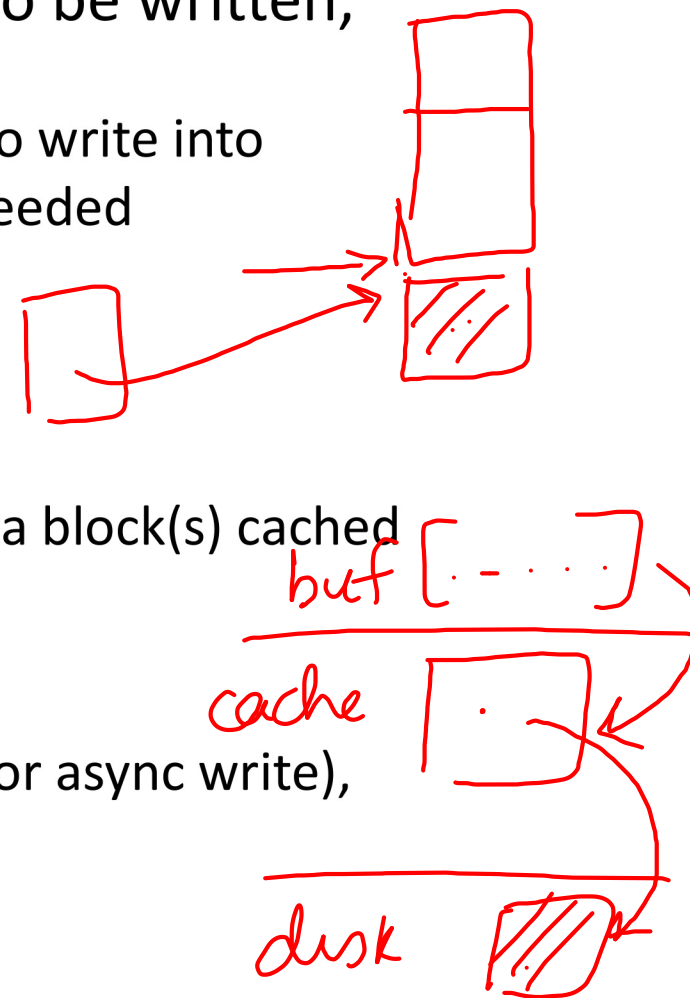
- Input is file descriptor, user memory to read into, number of bytes to read
 - Use file descriptor array index, access open file table entry, then inode
 - Based on offset, identify which data block(s) of file to read using inode information
 - Check if file data block(s) present in disk buffer cache
 - 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
 - Memory mapping a file avoids extra copy from disk buffer cache to user memory
 - 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

hit



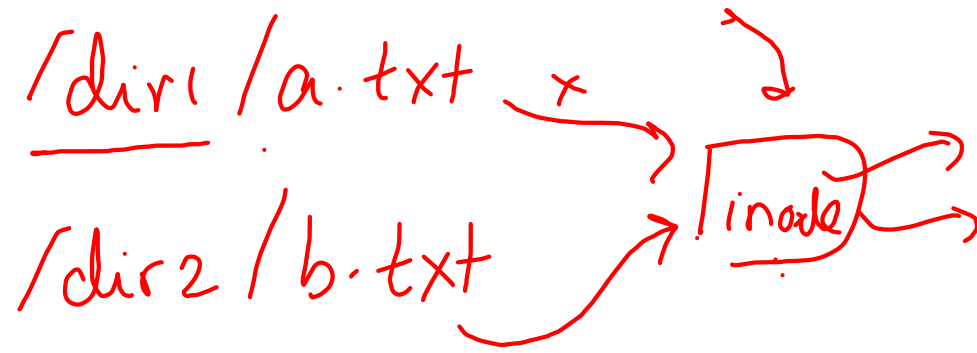
Write system call

- 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



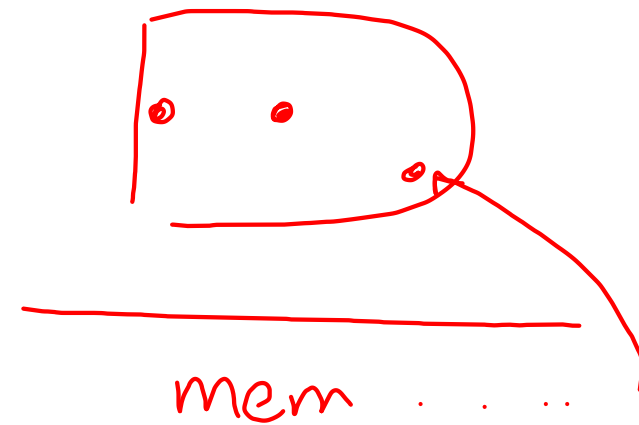
Linking and unlinking

ln

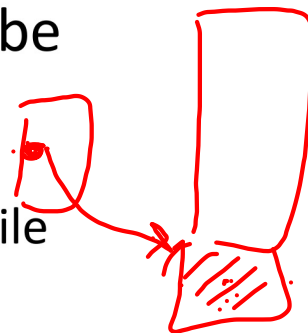


- Same file can be “linked” from different directories with different filenames using link system call
 - When file created, it is linked to its parent directory for first time
 - Subsequently, we can link to same file data from another directory also
- Hard linking: add entry in new directory, mapping 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” *a.txt* ~ *b.txt*
- 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

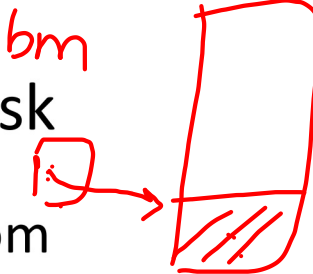


- Every system call updates multiple disk blocks
 - Example: when we append data to a file, we change data block, inode block, bitmap, ..
- All changes to disk blocks are first made in memory (disk buffer cache), then written to disk (synchronously or asynchronously)
 - Even metadata blocks (inode) are 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

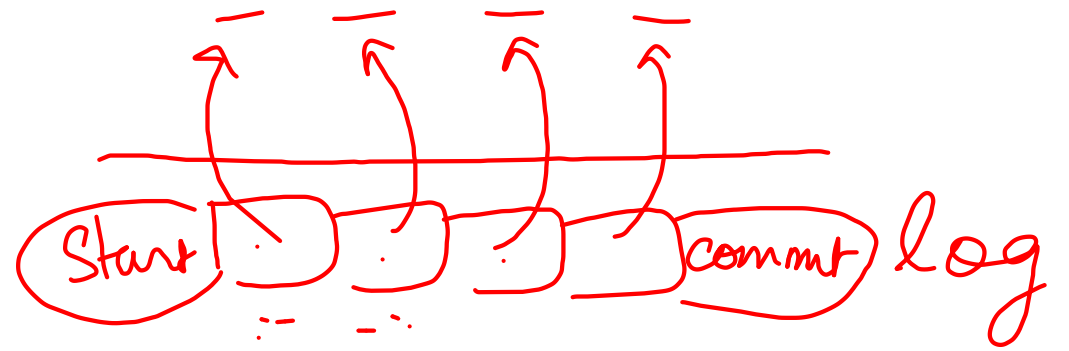


Filesystem checkers

- 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

Techniques for reliability

- Secondary storage devices fail sometimes, and lose/corrupt some/all of the data stored on them
 - Logging protects against power failures, but not data corruption on disk
- Techniques to protect data integrity in the face of disk failures
 - RAID (Redundant Array of Inexpensive Disks) uses multiple disks and distributes/mirrors data across disks
 - Checksum is additional bits stored along with data to detect data corruption (checksum recomputed and compared against original value when retrieving data) 
 - Error correcting codes add additional redundant bits to disk data to detect or recover from bit errors
 - Snapshotting or taking backups of disk data periodically, can rollback to previous snapshot in case of failures (copy-on-write technique used to make a copy of data when it changes, preserving both older and newer versions) 

Modern filesystems

- Filesystem is a way to organize data and metadata on secondary storage
 - Many ways of organizations → many different filesystems
- Modern filesystems differ on following parameters
 - Maximum size of files, maximum disk space used (limited by size of metadata structures)
 - Performance vs. reliability tradeoff (features like logging, checksums, coding, snapshots hurt performance but improve reliability)
 - Optimizations specific to technology used in secondary storage (traditional hard disks vs. SSD vs. non-volatile memory)
 - Optimizations specific to applications (storing general files vs. files of specific applications with specific characteristics)
 - Access to local vs. remote storage (local filesystems vs. network/distributed filesystems)
 - Support for compression, encryption, multi-level caching, and other advanced features
- No one size fits all, choice of filesystem depends on specific application

Virtual File System (VFS)

- Different filesystems can have different implementations of system calls
 - A filesystem using logging/journaling may write to log first
 - A different directory implementation (fixed size records vs linked list) will lead to a different lookup function
- How to write filesystem code in a modular manner?
 - Should be easy to change system call implementations and switch filesystems
- Solution: Virtual File System (VFS)
 - Defines a set of objects (files, directories, inodes) and operations to be performed on these objects (open a file, lookup filename in directory, ..) for various system calls
 - A specific filesystem implements these functions on VFS objects, provides pointers to the functions to be invoked by OS
- OS filesystem code is built in layers for modularity: VFS, filesystem implementation, disk buffer cache, device driver

VFS
—
FS impl
—
block
—
device driver

Summary

- In this lecture:
 - Implementation of file-related system calls
 - Logging for atomicity and crash consistency
 - VFS and layering for modularity
- Understand various system calls and how to use them (open, read, write, link, unlink, ..)
 - Many libraries available in all programming languages