Lecture 18: Unix Fast File System

CSL373

More UNIX files/inodes/directories

- How does system convert a name to an inode?
  - There is a routine called *namei* that does it
- How are symbolic links implemented?
  - A symbolic link is a file containing a filename
  - Macro substitution done by OS during operation
- What happens on ls, cd, cat? What happens on 'ls –F'?
- What about rm? Does it always delete a file?
  - NO. it decrements the reference count. If zero, free up file
- Will rm work for directories?
  - NO. because directory has a reference to itself (.)
  - Use a different command

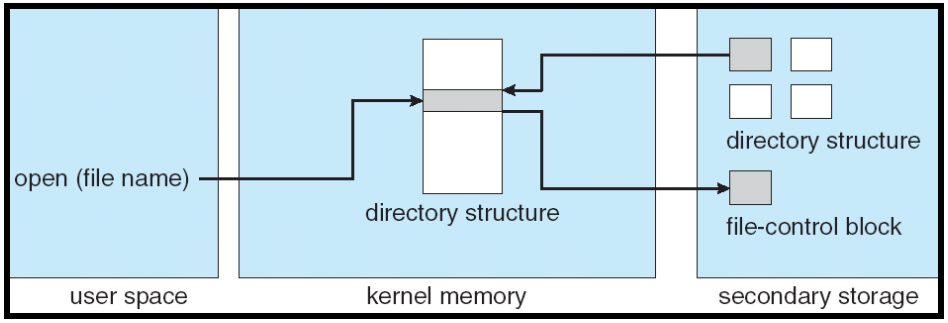
Where are inodes stored?

- In early UNIX and DOS/Windows' FAT file system, headers stored in special array in outermost cylinders
  - Header not stored near the data blocks. To read a small file, seek to get header, seek back to data.
  - Fixed size, set when disk is formatted. At formatting time, a fixed number of inodes were created (They were each given a unique number, called an "inumber")

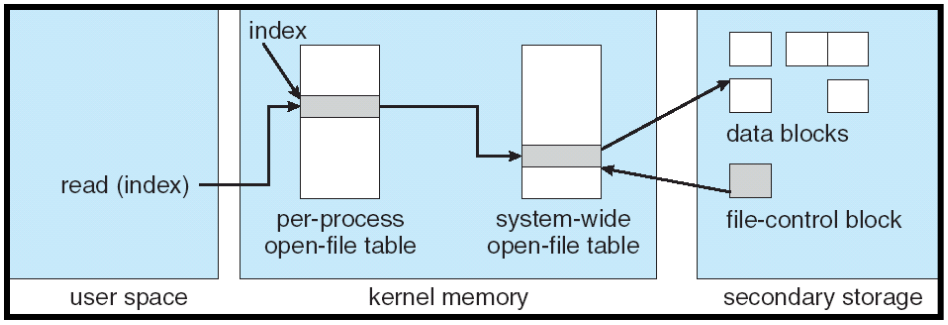
Where are inodes stored?

- Later versions of UNIX moved the header information to be closer to the data blocks
  - Often, inode for file stored in same "cylinder group" as parent directory of the file (makes an ls of that directory run fast).
  - Pros:
    - UNIX BSD 4.2 puts a portion of the file header array on each cylinder. For small directories, can fit all data, file headers, etc in same cylinder$\Rightarrow$no seeks!
    - File headers much smaller than whole block (a few hundred bytes), so multiple headers fetched from disk at same time
    - Reliability: whatever happens to the disk, you can find many of the files (even if directories disconnected)
  - Part of the Fast File System (FFS)
    - General optimization to avoid seeks

In-Memory File System Structures



- Open system call:
  - Resolves file name, finds file control block (inode)
  - Makes entries in per-process and system-wide tables
  - Returns index (called “file handle”) in open-file table



- Read/write system calls:
  - Use file handle to locate inode
  - Perform appropriate reads or writes

File System Caching

- Key Idea: Exploit locality by caching data in memory
  - Name translations: Mapping from paths→inodes
  - Disk blocks: Mapping from block address→disk content
- Buffer Cache: Memory used to cache kernel resources, including disk blocks and name translations
  - Can contain "dirty" blocks (blocks yet on disk)
- Replacement policy? LRU
  - Can afford overhead of timestamps for each disk block
  - Advantages:
    - Works very well for name translation
    - Works well in general as long as memory is big enough to accommodate a host's working set of files.
  - Disadvantages:
    - Fails when some application scans through file system, thereby flushing the cache with data used only once
    - Example: `find . -exec grep foo {} \;`
- Other Replacement Policies?
  - Some systems allow applications to request other policies
  - Example, 'Use Once':
    - File system can discard blocks as soon as they are used

File System Caching (con't)

- Cache Size: How much memory should the OS allocate to the buffer cache vs virtual memory?
  - Too much memory to the file system cache ⇒ won't be able to run many applications at once
  - Too little memory to file system cache ⇒ many applications may run slowly (disk caching not effective)
  - Solution: adjust boundary dynamically so that the disk access rates for paging and file access are balanced
- Read Ahead Prefetching: fetch sequential blocks early
  - Key Idea: exploit fact that most common file access is sequential by prefetching subsequent disk blocks ahead of current read request (if they are not already in memory)
  - Elevator algorithm can efficiently interleave groups of prefetches from concurrent applications
  - How much to prefetch?
    - Too many imposes delays on requests by other applications
    - Too few causes many seeks (and rotational delays) among concurrent file requests

File System Caching (con't)

- Delayed Writes: Writes to files not immediately sent out to disk
  - Instead, `write()` copies data from user space buffer to kernel buffer (in cache)
    - Enabled by presence of buffer cache: can leave written file blocks in cache for a while
    - If some other application tries to read data before written to disk, file system will read from cache
  - Flushed to disk periodically (e.g. in UNIX, every 30 sec)
  - Advantages:
    - Disk scheduler can efficiently order lots of requests
    - Disk allocation algorithm can be run with correct size value for a file
    - Some files need never get written to disk! (e..g temporary scratch files written /tmp often don't exist for 30 sec)
  - Disadvantages
    - What if system crashes before file has been written out?
    - Worse yet, what if system crashes before a directory file has been written out? (lose pointer to inode!)

How to make file system durable?

- Disk blocks contain Reed-Solomon error correcting codes (ECC) to deal with small defects in disk drive
  - Can allow recovery of data from small media defects
- Make sure writes survive in short term
  - Either abandon delayed writes or
  - use special, battery-backed RAM (called non-volatile RAM or NVRAM) for dirty blocks in buffer cache.
- Make sure that data survives in long term
  - Need to replicate! More than one copy of data!
  - Important element: independence of failure
    - Could put copies on one disk, but if disk head fails…
    - Could put copies on different disks, but if server fails…
    - Could put copies on different servers, but if building is struck by lightning….
    - Could put copies on servers in different continents…
- RAID: Redundant Arrays of Inexpensive Disks

Log Structured and Journaled File Systems

- Better reliability through use of log
  - All changes are treated as *transactions*
  - A transaction is *committed* once it is written to the log
    - Data forced to disk for reliability
    - Process can be accelerated with NVRAM
  - Although File system may not be updated immediately, data preserved in the log
- Difference between “Log Structured” and “Journaled”
  - In a Log Structured filesystem, data stays in log form
  - In a Journaled filesystem, Log used for recovery
- For Journaled system:
  - Log used to asynchronously update filesystem
    - Log entries removed after used
  - After crash:
    - Remaining transactions in the log performed (“Redo”)
    - Modifications done in way that can survive crashes
- Examples of Journaled File Systems:
  - Ext3 (Linux), XFS (Unix), etc.

Superblock

- When write a file, may need to allocate more inodes and disk blocks. The superblock keeps track of this data to help this process
- Superblock:
  - Size of file system
  - Number of free blocks in the file system
  - List of free blocks available in the file system
  - Index of next free block in free block list
  - Size of inode list
  - Number of free inodes in the file system
  - A cache of free inodes
  - The index of the next free inode in inode cache
- Superblock cached in memory, periodically flushed to disk. Also, replicated on disk for fault tolerance

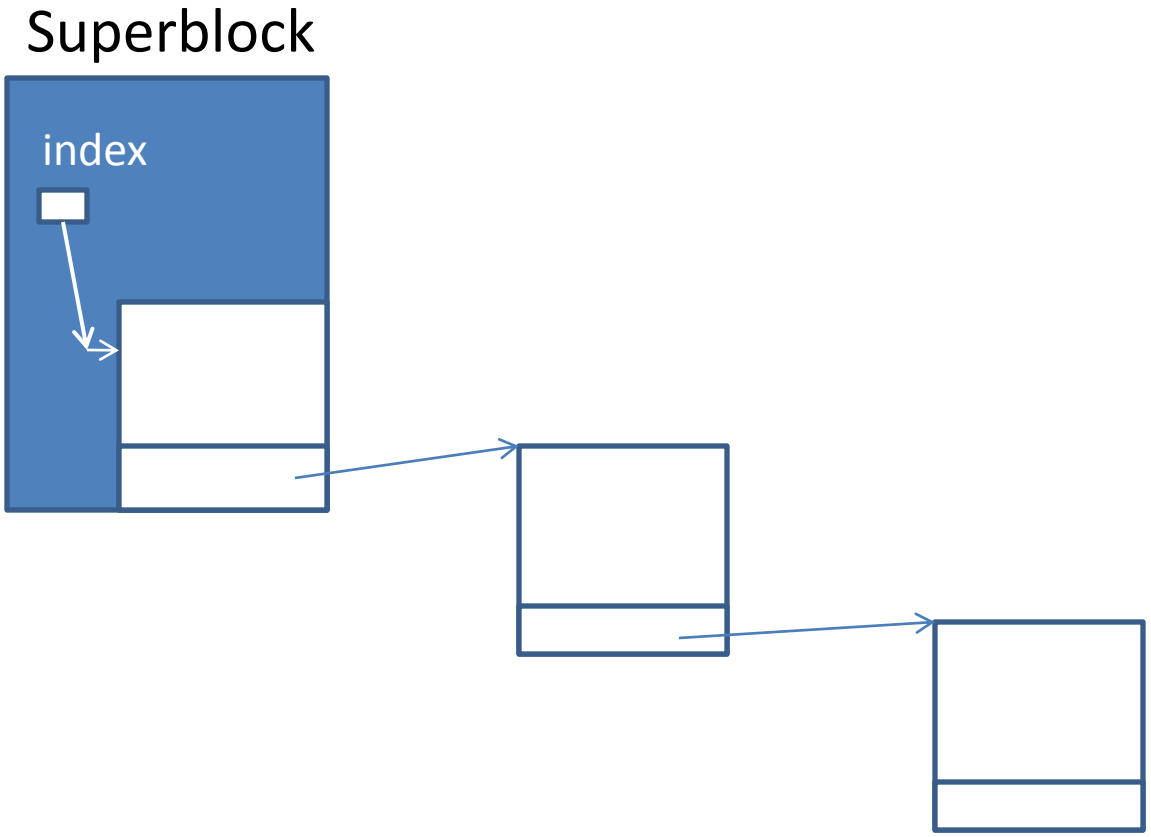
When OS wants to allocate an inode

- First look in inode cache
- Inode cache = stack of free inodes
- Index = top of stack
- When stack empty, search “inode list” for free inodes and fill up stack in superblock

To free an inode

- Put in superblock’s inode cache if there is room
- If not, don’t do anything much. Only check against the number where OS stopped looking for inodes the last time it filled the cache. Make this number the minimum of the freed inode number and the number already there

List of free disk blocks



When OS wants to allocate a disk block

- Check the superblock's block of free disk blocks.
- If there are at least two numbers, grab the one at the top and decrement the index of the next free block
- If there is only one number left, it contains the index of the next block in the disk block sequence. Copy this disk block into superblock's free disk block list, and use it as the free disk block

When OS wants to free a disk block

- If there is room in the superblock's disk block, push it on there.
- If not, write superblock's disk block into free block, then put index of newly free disk block in as first number in superblock's disk block

Why cache for inodes, and list for disk blocks

- Kernel can determine whether inode is free or not just by looking at it. But, cannot with disk block – any pattern is OK for disk blocks
- Easy to store lots of free disk block numbers in one disk block. But, inodes aren't large enough to store lots of inode numbers
- Users consume disk blocks faster than inodes. So, pauses to search for inodes aren't as bad as searching for disk blocks would be
- Inodes are small enough to read lots in a single disk operation. So, scanning lists of inodes is not so bad

Synchronization: Reads and Writes

- Read execute concurrently, read should either observe the entire write or none of the write.

- Reads can execute concurrently with no atomicity constraints.

Synchronization: How to implement

- Use reader-writer locks for each open file
- read_locks:
  - Acquire blocks until no other process has a write lock, then increments read lock count and returns.
  - Release decrements read lock count
- write_locks:
  - Acquire blocks until no other process has write or read lock, then sets write lock flag and returns
  - Release clears write lock flag
- Obtain read and write locks inside the kernel's system call handler
- File locks are based on inode instead of filename. Why?

Synchronization: Create, Open, Close, Delete

- If multiple processes have file open, and a process calls delete on that file, all processes must close the file before it is *actually* deleted.

Implementing Synchronization: Global File Table

- Have a global file table in addition to local file tables
- Global File Table: indexed by some global file id (e.g., inode index). Each entry has a reader/writer lock, a count of number of processes that have file open, and a bit that says whether or not to delete the file when the last process that has file open closes it. May have other data for other functionality.
- Local File Table: Indexed by open file id for that process. Has a pointer to the current position in the open file to start reading/writing

Sources of inefficiency: Wasted space and Wasted time

- Wasted time (basically due to scattering)
  - Inodes separated from files
  - Inodes in same directory scattered around in inode space
  - Disk blocks of one file scattered around
- Wasted space
  - Internal fragmentation due to block size

Wasted Time

- In traditional UNIX, block size = sector size
- When they went from 3BSD to 4.0BSD, they doubled the disk block size. This more than doubled the disk performance. Two factors:
  - Each block fetched twice as much data so amortized seek cost
  - File blocks were bigger so more files fit into direct section of the inode index
- But still pretty bad. When file system first created, got transfer rates of up to 175 KBps. After a few weeks, deteriorated down to 30KBps. What is worse is only about 4% of max disk throughput. So, obvious fix is to make the block size even bigger

Wasted Space

- Problem is bad, because most files are small.

Space Used (MB)	Percent Waste	Organization
775.2	0.0	Data only, no separation between files
828,7	6.9	Data+inodes, 512 byte block
866.5	11.8	Data+inodes, 1024 byte block
948.5	22.4	Data+inodes, 2048 byte block
1128.3	45.6	Data+inodes, 4096 byte block

Cylinder Groups (BSD 4.2)

- Cylinder group=set of adjacent cylinders
- A filesystem consists of a set of cylinder blocks
- Each cylinder group has
  - A redundant copy of the super block
  - Space for inodes
    - Default Policy: Allocate 1 inode per 2048 bytes of space in cylinder group
  - Bitmap describing available blocks in the cylinder group
- Basic Idea: Put related info together in the same cylinder group, and unrelated info apart in different cylinder groups. Use a bunch of heuristics

Cylinder Groups Heuristics

- Try to put all inodes for a given directory in the same cylinder group
- Try to put blocks for one file adjacent in the cylinder group. The bitmap as a storage device makes it easier to find adjacent groups of blocks.
- For long files, redirect blocks to a new cylinder group every megabyte. This spreads stuff out over the disk at a large enough granularity to amortize the seek time

Design choices

- Keep a free space reserve (5 to 10 percent)
  - Necessary for good performance
  - Once above reserve, only supervisor can allocate disk blocks
  - If disk almost completely full, allocation scheme cannot keep related data together and allocation scheme degenerates to random
- Increased block size (4096 bytes)
  - Helps read bandwidth and write bandwidth for big files
  - But wastes space for small files. Solution: Introduce concept of disk block fragment

Disk Block Fragments

- Each disk block can be chopped up into 2,4, or 8 fragments
- Each file contains at most one fragment wiich holds the last part of data in the file.
- So, if have 8 small files they together occupy only one disk block. Can also allocate larger fragments if the end of the file is larger than one-eighth of the disk block
- The bitmap is laid out at the granularity of fragments
- When increase the size of the file, may need to copy out the last fragment if the size gets too big
- Bottom line: read bandwidth up to 43% of peak disk transfer rate for large files

Disk block cache

- OS maintains a cache of disk blocks in main memory
- Cache replacement policy:
  - LRU is easy to implement (compare with VM) but not scan-resistant (sequential access)
  - Read-ahead helps with sequential access
- Can we use the file system as a backing store for VM? Can run into double caching