CSE 120
Principles of Operating Systems

Spring 2018

Lecture 13: File System Implementation

Geoffrey M. Voelker

File Systems

- First we'll discuss properties of physical disks
  - Structure
  - Performance
- Then how file systems support users and programs
  - Files
  - Directories
  - Sharing
  - Protection
- End with how file systems are implemented
  - File System Data Structures
  - File Buffer Cache
  - Read Ahead

File System Layout

- We start with an empty disk

- Goal for the file system is to manage the disk space to implement the file and directory abstractions that are so convenient for programs and users

Key Questions

- How do we keep track of blocks used by a file?
- Where do we store metadata information?
- How do we (really) do path name translation?
- How do we implement common file operations?
- How can we cache data to improve performance?

- Our discussion will be Unix-oriented
  - Other file systems face same challenges, with analogous approaches and data structures for solving them

File System Layout

How do file systems use the disk to store files?

- File systems define a block size (e.g., 4KB)
  - Disk space is allocated in granularity of blocks
- A “Master Block” determines location of root directory
  - Always at a well-known disk location
  - Often replicated across disk for reliability
- A free map determines which blocks are free, allocated
  - Usually a bitmap, one bit per block on the disk
  - Also stored on disk, cached in memory for performance
- Remaining disk blocks used to store files (and dirs)
  - There are many ways to do this

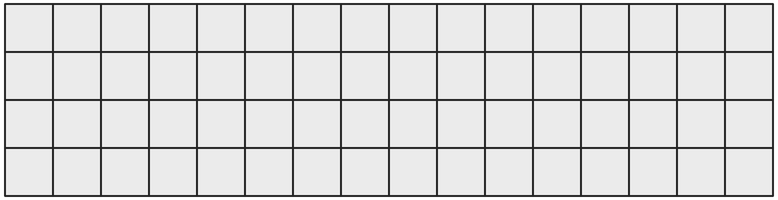
File System Layout

How do file systems use the disk to store files?

- File systems define a block size (e.g., 4KB)
  - Disk space is allocated in granularity of blocks
- A "Master Block" determines location of root directory
  - Always at a well-known disk location
  - Often replicated across disk for reliability
- A free map determines which blocks are free, allocated
  - Usually a bitmap, one bit per block on the disk
  - Also stored on disk, cached in memory for performance
- Remaining disk blocks used to store files (and dirs)
  - There are many ways to do this

File System Layout

- Partition it into fixed-size file system blocks



- Typically 4KB in size
  - Block size set when file system is formatted
- Independent of disk physical sector size
  - Sector 512 bytes, file system will use 8 sectors/block

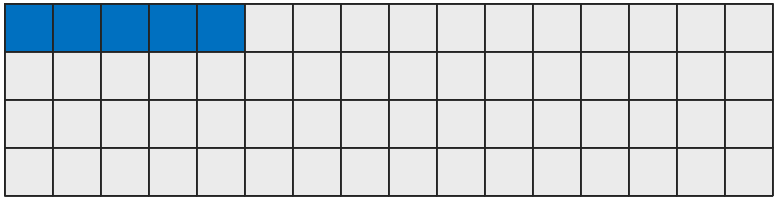
File System Layout

- Files span multiple disk blocks
  - 2MB file uses 2*1024*1024/4096 = 512 blocks (4KB block size)
- A small file still uses an entire block
  - A file of size 4001 bytes uses one block
  - What kind of fragmentation is this, internal or external?

- Challenge: How do we keep track of all of the blocks used by one file?

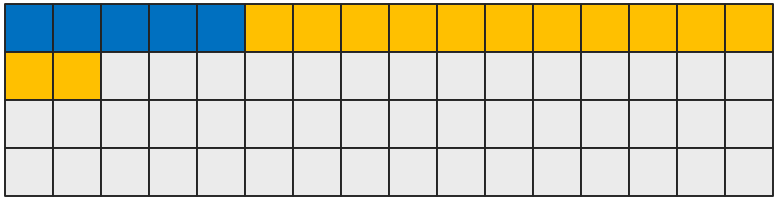
Contiguous Layout

- Can layout file blocks contiguously



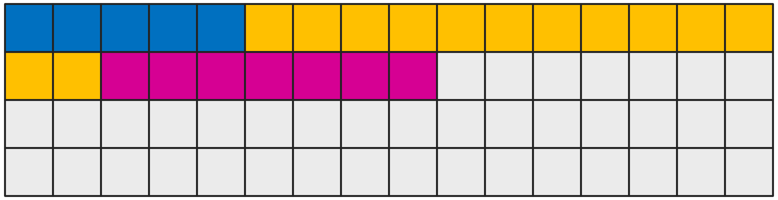
Contiguous Layout

- Can layout file blocks contiguously



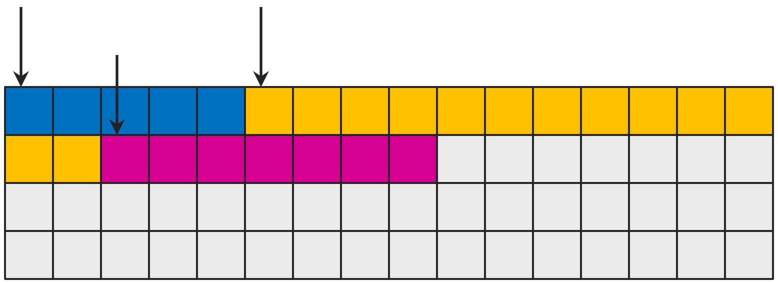
Contiguous Layout

- Can layout file blocks contiguously



Contiguous Layout

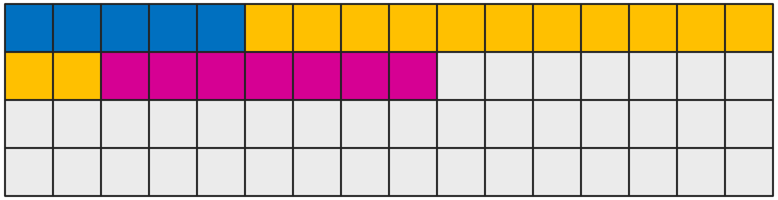
- Simple to keep track of where a file's blocks are



- Directory stores a pointer to the first block
  - All others are a simple offset from the first
  - Makes random access also straightforward
- Enables fast sequential access to disk for reads/writes
- But there are multiple disadvantages

Contiguous Layout

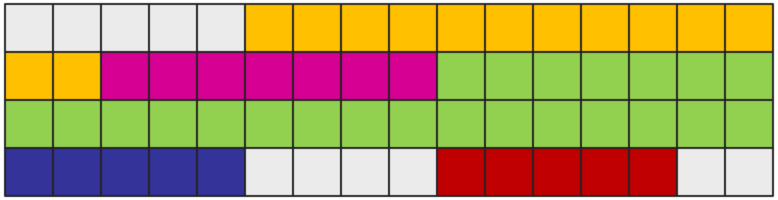
- Difficult to grow a file once it has been written



- If the blue or orange files need to grow, we’re stuck

Contiguous Layout

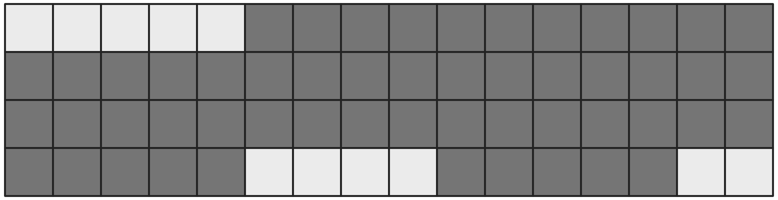
- As files are created and deleted, gaps will occur



- If we need to store a file using 8 blocks, we're stuck
  - What kind of fragmentation is this, internal or external?
  - What would be one method for rearranging?

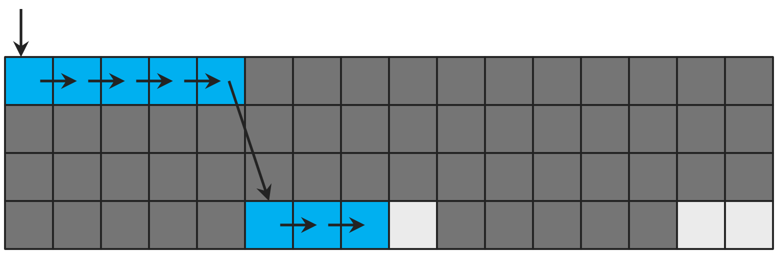
Linked Layout

- Need to store a file with 8 blocks into the “gaps”



Linked Layout

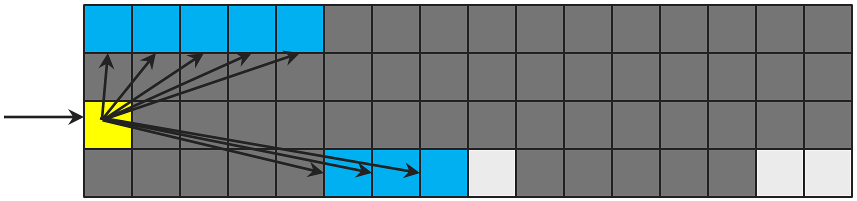
- Another option is to link each block to the next
  - Essentially a linked list on disk for each file



- Directory still just stores pointer to first block of file
- Fragmentation no longer a problem, can fill in gaps
- Random access now expensive
  - Need to traverse pointers to access a random block
  - Potentially many disk reads just to get to desired block

Indexed Layout

- Indexed layouts use a special block (index block) to store pointers to the data blocks



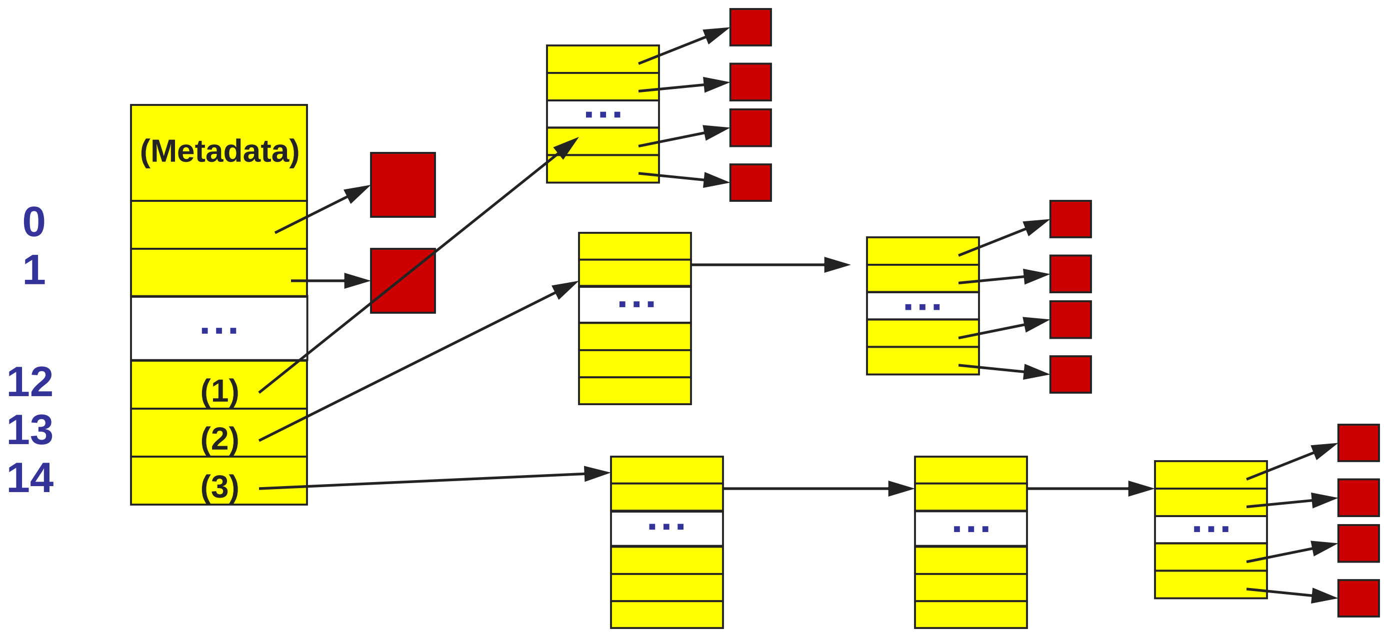
- Directory points to the index block
- Still solves fragmentation problem (can fill in gaps)
- Also solves random access problem
  - After reading the index block, know the locations of all blocks
- For large files, need multiple index blocks

Disk Layout Summary

- Files span multiple disk blocks
- How do you find all of the blocks for a file?
  1. Contiguous allocation
     - » Like memory
     - » Fast, simplifies directory access
     - » Inflexible, causes fragmentation, needs compaction
  2. Linked structure
     - » Each block points to the next, directory points to the first
     - » Good for sequential access, bad for all others
  3. Indexed structure (indirection, hierarchy)
     - » An “index block” contains pointers to many other blocks
     - » Handles random better, still good for sequential
     - » May need multiple index blocks (linked together)

Unix Inodes

- Unix inodes implement an indexed structure for files
- Each inode contains 15 block pointers
  - First 12 are direct blocks (e.g., 4 KB blocks)
  - Then single, double, and triple indirect



File Metadata

- Unix inodes also store all of the metadata for a file
- File size
  - In bytes (actual file size) and blocks (data blocks allocated)
- User & group of file owner
- Protection bits
  - User/group/other, read/write/execute
- Reference count
  - How many directory entries point to this inode
- Timestamps
  - Created, modified, last accessed, any change
- "ls –l" reads this info from the inode (syscall: stat())

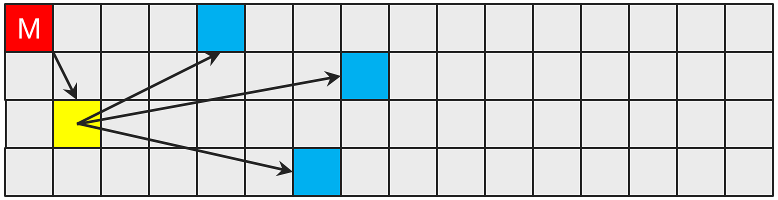
File System Layout

How do file systems use the disk to store files?

- File systems define a block size (e.g., 4KB)
  - Disk space is allocated in granularity of blocks
- A “Master Block” determines location of root directory
  - Always at a well-known disk location
  - Often replicated across disk for reliability
- A free map determines which blocks are free, allocated
  - Usually a bitmap, one bit per block on the disk
  - Also stored on disk, cached in memory for performance
- Remaining disk blocks used to store files (and dirs)
  - There are many ways to do this

Master Block (Superblock)

- "/" is the directory that is the root of the file system
- How do we find the inode for "/"?



- The master block M stores a pointer to the inode of "/"
  - The inode for "/" has pointers to all of the blocks storing the directory entries for "/"
- It is the basis for translating all path names
- It is at a fixed, pre-defined location on disk
  - Replicated deterministically across the FS for redundancy

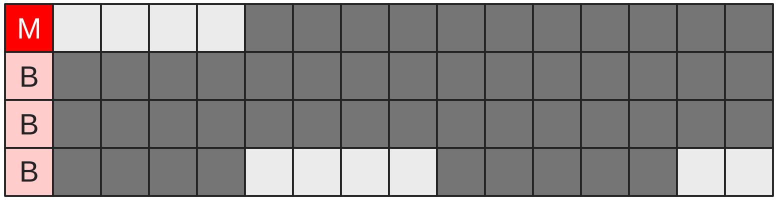
File System Layout

How do file systems use the disk to store files?

- File systems define a block size (e.g., 4KB)
  - Disk space is allocated in granularity of blocks
- A “Master Block” determines location of root directory
  - Always at a well-known disk location
  - Often replicated across disk for reliability
- A free map determines which blocks are free, allocated
  - Usually a bitmap, one bit per block on the disk
  - Also stored on disk, cached in memory for performance
- Remaining disk blocks used to store files (and dirs)
  - There are many ways to do this

Block Allocation

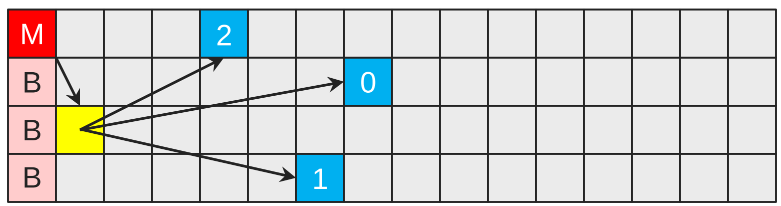
- The file system needs to keep track of which blocks have been allocated and which are free



- Free map blocks B store a bitmap, one bit per block
  - Bit is set → block is allocated
  - One bitmap for data blocks
  - Another for inode blocks

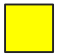
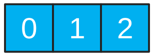
Types of Blocks on Disk

- Four basic kinds of blocks on disk
  - Only data blocks store file data and directory data



- Master block 

- Bitmap blocks 

- Inode blocks 
- Data blocks 

Unix Inodes != Directories

- Unix inodes are not directories
  - Inodes describe where on the disk the blocks of a file are
- Directories are files, so inodes also describe where the blocks for directories are placed on the disk
  - Every directory and file on disk has an associated inode for it

Path Name Translation (v2)

- Directory entries map file names to inodes
  - To open “/one”, use master block to find inode for “/” on disk
  - Open “/”, look for entry for “one”
  - This entry gives the disk block number for the inode for “one”
  - Read the inode for “one” into memory
  - The inode says where first data block is on disk
  - Read that block into memory to access the data in the file

Symbolic (Soft) Links

- It is convenient to be able to create aliases in the FS
  - Have a file be referred to by multiple names
- Soft links are the most familiar form in Unix

  % ln –s *file alias* (ln –s /a/b/c /tmp/softlink)
  - Syscall: symlink()
- Soft links create aliases via path name translation
  - Path name translation starts again when hitting a soft link
  - /tmp/softlink → /a/b/c
- Implemented simply by storing the alias as a string in a file and marking the inode as a soft link
  - FS reads the path alias from the file and restarts translation

Hard Links

- Hard links are another form of aliasing
  % ln *file alias* (ln /a/b/c /tmp/hardlink)
  - Syscall: link()
- Hard links create aliases via inode pointers in dirs
  - Recall that a directory entry maps a name to an inode
  - Creating a hard link adds another directory entry mapping the new name to the same inode as the old name
  - It adds a new pointer, or link, to the inode
  - Reference count in the inode is also incremented
- The "." and ".." names are hard links to directories
  - /a/b/c and /a/b/c/. point to the same inode

Create

- Creating a file “new” is relatively straightforward
- Allocate an inode
  - Initialize the metadata (owner, protection, timestamps, …)
  - Update inode bitmap
- Allocate a directory entry in the directory for the file
  - Entry maps “new” to the inode allocated for “new”
- When process starts writing to file, allocate data blocks
  - Update inode to point to data blocks allocated
  - Update data block bitmap
  - Continue to allocate data blocks on demand
    - Preallocating blocks in “extents” helps keep blocks contiguous

Rename

- One way to rename a file is to simply create a new one with the new name, copy the contents, and delete the old file
  - Method used in original version of Unix (test/mv.c in Nachos)
- More efficient to implement in FS

  % mv old new

  - Syscall: rename()
- Rename creates a new directory entry with the new name that points (links) to the same inode as the old
  - Then it deletes the entry directory for the old name
  - Only directories are modified, file and inode stay the same

Delete

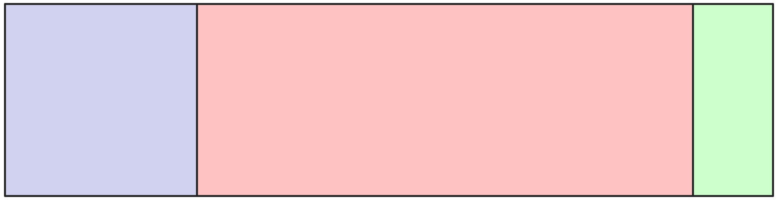
- Deleting a file has a few steps
  - Remove the directory entry for the name being deleted
    - Hence the syscall name unlink()
  - Decrement the reference count in the inode
  - If the file still has links to it, nothing else happens
- If there are no remaining links
  - Free up the data blocks (update the data block bitmap)
  - Free up the inode blocks (update the inode bitmap)
  - Block data is not erased
- If the file is still open in any process, the directory entry is removed but the file blocks are not
  - Until the last process with the file open finally closes it

Partitions

- What if we want multiple file systems on one disk?

Partitions

- What if we want multiple file systems on one disk?



- Split up physical disk into multiple partitions
- Each partition has an entire file system inside of it
  - Master block, bitmaps, etc.
- How do we link them together into one name space?

Mounting File Systems

- Mounting is the mechanism used to piece together multiple file systems into a single global name space
- One file system is mounted as “/” (root)
- Other file systems attached at mount points
  - An empty directory in file system, e.g., /home
  - Mounting the “home” file system attaches the root for “home” to /home in the name space
  - Opening “/home/voelker/file” starts path name translation in the “root” file system, continues in the “home” file system when crossing the mount point
- Mostly invisible to users and processes
  - Some exceptions (e.g., cannot hard link across file systems)

File Buffer Cache

- Applications exhibit significant locality for reading and writing files
- Idea: Cache file blocks in memory to capture locality
  - Called the file buffer cache
  - Cache is system wide, used and shared by all processes
  - Reading from the cache makes a disk perform like memory
  - Even a small cache can be very effective
- Issues
  - The file buffer cache competes with VM
    - » Tradeoff: More physical memory for file cache, less for VM
  - Like VM, it has limited size
  - Need replacement algorithms again (LRU usually used)

Caching Writes

- On a write, some applications assume that data makes it through the buffer cache and onto the disk
  - As a result, writes are often slow even with caching
- OSes typically do write back caching
  - Maintain a queue of uncommitted blocks
  - Periodically flush the queue to disk (30 second threshold)
  - If blocks changed many times in 30 secs, only need one I/O
  - If blocks deleted before 30 secs (e.g., /tmp), no I/Os needed
- Unreliable, but practical
  - On a crash, all writes within last 30 secs are lost
  - Modern OSes do this by default; too slow otherwise
  - System calls (Unix: fsync) enable apps to force data to disk

Read Ahead

- Many file systems implement "read ahead"
  - FS predicts that the process will request next block
  - FS goes ahead and requests it from the disk
  - Can happen while the process is computing on previous block
    - Overlap I/O with execution
  - When the process requests block, it will be in cache
  - Compliments the disk cache, which also is doing read ahead
- For sequentially accessed files can be a big win
  - Unless blocks for the file are scattered across the disk
  - File systems try to prevent that, though (during allocation)

Next time...

- Read Chapters 37, 39, 40