Last Updated: 2-10-23

More File Systems

# Building A File System

* All we have is an array of fixed-sized data blocks
* There are three capabilities we need on top of the data blocks

1. Track which blocks store each file's contents (they can be spread over a disk!)
   1. Inodes holds metadata & disk indexes; stat() and fstat() used to access inode
   2. Why are the file blocks nonconcurrent?
      1. Pros: shrinking/growing a file is now easy - any free block is as good as any other free block in the array, so no need to search through variable size allocations and free regions to find an area of correct size
      2. Cons: more complicated bookkeeping in file system (inode trees)
2. Track which blocks are free and which are available
   1. Not that hard - use one bit per block (free map)
3. Creating a directory hierarchy and mapping from file names to their bookkeeping information

* Finding the right inode
  + Each inode has is assigned a number giving its position in the array; this gives us the disk block number
  + Number of bits in inode numbers dictate size of array, which dictates maximum number of files possible; I could run out of inodes before I run out of data blocks

## Directories

* + - A directory is just a sequence of entries; each entry is a pair: name and (inode) number (e.g., hello.txt 55280)
    - *Directories are just a specialized type of file*
      * They have inodes and data blocks as well; it’s just that the contents of the data blocks is the sequence of filename and number pairs

# Inside I/O Calls

## What’s inside a call to open()

int fd = open(“/home/sun/hello.txt”)

1. Inode number for the “/” directory is known ahead of time
2. Open and read the “/” directory to find the entry with the name “home”
3. Inode number for “/home” (in this instance) is 8
4. Open and read the “/home” directory to find entry with the name “sun”
5. Inode number for “/home/sun” (in this instance) is 15
6. Open and read the “/home/sun” directory to find entry with the name “hello.txt”
7. Inode number for “/home/sun/hello.txt” (in this instance) is 16
   * Lots of reading from the disk, so it seems like it would be slow, right?
     + **Recall:** [Inode table store](https://docs.google.com/document/u/0/d/14r-w8vf6TndK5NJKwDY6J91SwYeYmUW09mM8sZheP8g/edit)s or cache inodes in memory so that they are faster to access
       - When we open files, we must check that the user has the necessary permissions by checking the inode - *this includes directories* 
         * Write - can add/delete files from a directory
         * Read - can look at file names inside a directory
         * Execute - can *traverse* the directory

* What’s inside a read()/write()?
  + **Recall:** there lies an extra layer of buffering between read/write system calls & file contents on a disk

1. File is opened - new or existing inode table entry is copied into memory to make access faster
2. read() completes - copy some of the contents of disk blocks into memory and leave them in memory - this portion of memory that is reserved for data block contents under kernel control is given the (very creative) name: **the buffer cache**
3. write() completes - modify data in the buffer cache
4. OS flushes buffers - synchronize data blocks on disk
   1. Happens when you call fsync() or periodically every ~15 seconds

# Directory Operations

* Instead of using open(), read(), write() & close(), for directories, use opendir(), readdir() & closedir()
  + No directly writing to directories - other file operations such as open() creates and modifies directory contents
* DIR \*opendir(char\* dirname);
  + Instead of FILE\*, we use DIR \*
  + Note: Cannot assume the order of entries inside a directory - “Implementation Dependent”
* struct dirent \*readir(DIR\* dirp);
  + Returns next directory entry
  + Returns NULL if at end of directory or error
  + Repeat calls to this function, much like strtok()
  + Directory contains at least name and inode number
* int closedir(DIR\* dirp)
* void rewinddir(DIR\* dirp)
  + Like a “seek” back to the beginning

# File Operations - What Happens

* Make a new file
  + Allocate a new inode
  + Add a directory entry referring to the inode so that it is reachable
* Modify a file
  + Allocate/free data blocks to match new file size
  + Change content in relevant data blocks
* Renaming a file
  + Modify file’s directory entry
  + Don’t need to worry about inode or data blocks
* Moving a file
  + Remove file’s entry from source directory and add a new entry to destination directory with the same inode

# Links

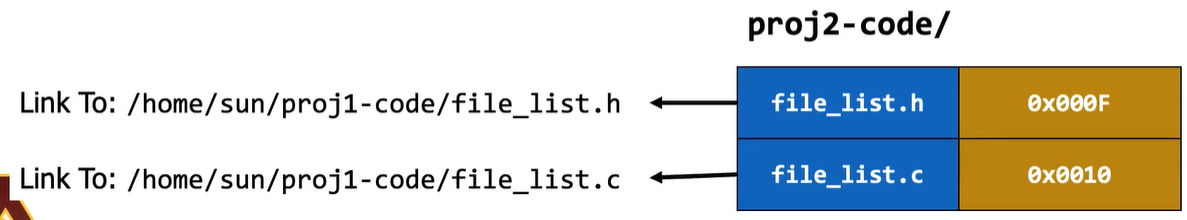
* **Recall**: A files metadata and contents are (inode & disk blocks) are distinct from its name and location within the file system tree (directory entry)
* This allows us to create additional edges in the file system “tree”, turning it into a DAG
* Two kinds of links can be created

## Hard Links

* + - A **hard link** is a pointer from a directory entry (a file name and location within the directory hierarchy) to an inode
    - We’ve seen this before: we lookup a file’s inode from a dir. ent (e.g., in a call to open())
    - An inode can be pointed at by *more than one* directory entry
      * Inode maintains reference count, file is only deleted when count is 0
    - When you make a file, you create a hard link; deleting a file deletes the hard link

## Soft (Symbolic) Links

* + - More like the concept of a shortcut
    - Independent directory entry that points to another file by name (not by inode of target
      * If target is deleted, link remains but is invalid
    - Inode is unaware of any symbolic links pointing to it
    - Implementation: file with path to another file as its contents, but indicates symbolic link status in inode for special treatment



* Linking Operations in Terminal
  + Create links using “ln <directory path to file>”, include a “-s” flag for symbolic links
* Related System Calls
  + link() creates a hard link
  + unlink() to remove file
  + rename() to rename or relocate a file, guaranteed to be [atomic](https://docs.google.com/document/d/14r-w8vf6TndK5NJKwDY6J91SwYeYmUW09mM8sZheP8g/edit#heading=h.72ot6krav0i5)