CS 214 / 2021-02-24

```
Class business:
        Asst 2
                Due Feb 25
        Project 1
                Declare partnership by March 3
                Due March 12
```

Inodes and directory entries

```
Note: We are going to talk about the Unix file system; not all file systems
work the same way, but most file systems have analogous concepts.

What is a file?
- in Unix, a sequence of bytes that is stored somewhere and we can refer to
  by name

How do we store a file on disk?
- naive approach: write data contiguously to disk (similar to heap)
        - problem: files often grow or shrink
                -> may not be room for a file to grow
                -> shrinking a file may lead to gaps between files
        - files can be deleted, this will lead to gaps
        -> disk-based systems do not work this way
                -> some tape-systems do, but these are rare nowadays

-> instead, we break files into parts ("blocks")
        -> each block is the same size
        -> blocks can be packed into the disk (like an array)
        -> large files will be made of multiple blocks
        -> now we need a way to indicate which blocks make up a file

In Unix, we distinguish data and metadata
        data - actual content of the file
        metadata - information about the file
                - how big it is
                - when it was created/modified/accessed
                - who has access
                - which blocks are used to store the data

-> the metadata for a file is stored in an "inode" ("i-node")
        -> Each inode has a unique number (ID)
        -> Each inode is the same size

Problem: how can we keep track of a variable amount of data using a fixed
amount of metadata?
        -> we want the maximum file size to be very large
        -> we don't want the inode to be too big, because most files are small
        -> we want random-access to file contents

we could use a linked list, where each block refers to the next block in the
file
        - file if we only read sequentially
        - but if we want to append to a file made of 1000 blocks, we have to walk
          through every block to find the end

We could just list all the blocks in the inode
        -> this wastes space for small files
                -> not helpful if we want to allow large files, but expect most files
                   to be small

Unix file system uses a system of indirect accesses

For some N, the first N blocks are directly referenced in the inode
Next, we have indirect references
        The inode refers to a block that contains block references

For example, let's say the inode contains 25 direct references
let's also say that a data block can hold 100 block IDs

-> the first 25 blocks are referred to directly by the inode
-> the next 100 blocks are given by the single indirect node
                1 reference in inode gives us 100 indirect references
-> the next 10,000 blocks are given by the double indirect node
                1 reference in inode gives us 100 blocks, each of which points to 100
                        blocks
-> the next 1,000,000 blocks are given by the triple indirect node
                1 reference in inode
                        -> 100 references in block
                                -> each leads to 100 references
                                        -> each pointing to 100 blocks

Summary: 28 entries in inode allows us to reference up to 1,010,125 blocks

The block's number in the file tells us how to find it
            0..       24 - direct reference
           25..      124 - single indirect (index 25 in inode)
          125..   10,124 - double indirect (index 26 in inode)
       10,125..1,010,124 - triple indirect (index 27 in inode)

Note: 25 and 100 are just numbers I chose for this example
        both will be larger for a real file system

-> fixed size for inode
-> large maximum size for file
-> minimal space usage for small files
-> constant-time access for blocks
        -> later blocks in the file take longer to find, but we are limited to 3

E.g., to examine block 1,000,000, we only need to access the inode and three
        indirection blocks to get to the block
For a linked list, we would need to go through all 1,000,000 preceding blocks

This is a good balance between speed and flexibility
        -> no overhead for small files
        -> small overhead for large files

        -> essentially a lopsided, N-way tree

Why only triple indirection?
        No technical limitation here; we need to have a limit because inodes are
                fixed size
        This means we do have a maximum file size
                -> on modern file systems, the maximum file size is very, very big

Takeaway: fixed size metadata
        allow for very large files
        minimal overhead for small files
        fast random access



Aside: Non-sequential file access
        if we open a file in append mode (O_APPEND), then we will start writing
                from the end of the file
        lseek() lets us move the file cursor anywhere in the file
                i.e., we can skip ahead or go backward


Also of note:
        Unix-like filesystems use inodes to reference files
                we can refer to a file by its inode ID
                (the file system keeps track of how to find an inode given its ID)

        Notably, the file name is not part of the file metadata!
                -> file names are part of the directory listing


What is a directory?
        A special file that contains directory entries
                each entry contains some information, including
                        - the name of the file
                        - what type of file it is (regular file, directory, other)
                                not related to extension, or the type of data in the file
                                programs, source code, text files, photos, etc. are all regular files
                        - its inode ID

The file system tracks the root directory /

        / contains its subdirectories & files
                each subdirectory contains its subdirectories and files

If we have a path like /user/foo/homework/hw1.txt, we can find its inode ID
        start from /
        look up "user" in /
        look up "foo" in /user
        look up "homework" in /usr/foo
        look up "hw1.txt" in /usr/foo/homework
        get inode ID

We say that the name (or path) of a file links to its inode ID
        the inode ID is the "true" name of the file
        the path is the user-friendly alias

However:
        Not every file is linked from the directory structure
        Some files can be linked more than once
                -> we can have multiple names for the same file

        A file's inode tracks how many times it is linked
                (i.e., how many names it has)
        When we use rm to delete a file, we unlink the name from the file
                if a file has no links, the file system deletes it


Use ln to create additional names for a file (sometimes called a hard link)

        ln existing_file new_name

        existing_file and new_name will both refer to the same inode
        they are indistinguishable

        If I then rename existing_file, the new name will still refer to the same
                inode as new_name

Contrast with symbolic links

        ln -s existing_name new_name

        new_name is a new file that refers to the name "existing_name"

        If I rename existing_name, the link breaks
        If I then create a new file called existing_name, then new_name will refer to that
```

Stat

```
References (on iLab)
        man 2 stat
        man 7 inode

How can we get information about a file?

stat/fstat/lstat - gives us data from the inode

given a file name, is this file a regular file or a directory?
        how big is it?
        what is its creation/modification/access date?
        who can access it?
```

```c
#include <sys/stat.h>

int isdir(char *name) {
        struct stat data;

        int err = stat(name, &data);

        // should confirm err == 0
        if (err) {
                perror(name);  // print error message
                return 0;
        }

        if (S_ISDIR(data.st_mode)) {
                // S_ISDIR macro is true if the st_mode says the file is a directory
                // S_ISREG macro is true if the st_mode says the file is a regular file

                return 1;
        }

        return 0;
}
```

Working with directories

Next time: opendir / readdir