Background

- What is a process?
- What is an address space?
- What is a system call?

The File System API

FS System Calls

If we focus on Linux, there are many system calls (~300), but only some of them specifically relate to storage. The system calls discussed in OSTEP Chapter 39 are enumerated below, and they are the ones that you will likely use most in this class. You should be familiar with all of them: both how to use them (or how to look up their usage using Unix man pages) and how they affect the state of the storage system's data structures. Memorization is less important than thinking about why the interface is designed the way it is.

- open/creat
- close
- read
- write
- lseek
- pread/pwrite
- fsync
- rename
- stat/fstat
- unlink
- mkdir
- readdir
- rmdir
- unlink
- mount
- umount

Action items

- How do you look up the interface for any one of these system calls?
- How can you tell what system calls are performed when you run a program from the command line?
Names

There are multiple ways to refer to files, each with their own advantages and disadvantages. You should be familiar with the types of identifiers that are passed to each FS-related system call, and why that particular identifier is used. What are the uses, advantages, and disadvantages for each of the following types of identifiers? Thoughts: Uniqueness? Easy to remember? Relationship to links?

- inode number
- path
- file descriptor

files (the data structure)

Each process contains a private table that maps file descriptors (integer identifiers) to file data structures. We should try to be precise when we use the term file: colloquially it has a meaning that is similar-to-but-very-different-from the file data structure that is part of the file system API in the Unix kernel. Unfortunately this is difficult, so context should be helpful when determining what is meant.

- What field(s) are stored in the file data structure?
- What system calls alter those fields?
- What is the relationship between the file data structure, a process, and the notion of a file that we colloquially use (a named unit of persistent storage)?
- Thinking forward to next class: what is the granularity of access to a file? What is the granularity of access to a block device like a HDD? What challenges might arise as a result of this mismatch?

directories

Directories do not store "data" in the typical sense. Directories are a particular type of file that contains a mapping from names to inode numbers.

- All directories contain two files by default: . and .. What are these files and what is their purpose?
- rmdir deletes a directory. What are the necessary preconditions for the successful deletion of a directory? Why do you think this decision was made?

open

- Which type of the three identifiers (in the Names section above) do you pass as an argument to open?
- What type of identifier does open return?
- There are three types of identifiers listed above... why isn't the third type of name used?
- What is the difference between open and creat? Given this relationship, how might you implement creat?

links

- What is meant by a hard link and a symbolic link?
- Reference counting is an important concept in file systems (and other systems for that matter). Which type of link (hard or symbolic) increases an inode's reference count?
  - What system call lowers an inode's reference count?
- What is meant by a dangling reference?
- In what scenarios would you use a hard link, and in what scenarios would you use a symbolic link?
- Name one common use of symbolic links.
File Systems and Trees

- What is a "tree" and why is it important that the file system namespace is a "tree"?
- How does Unix use a tree to let us combine multiple file systems?
- What is a "path lookup" and what are the steps involved in taking an absolute path and opening a file? (This isn't described in detail in the assigned readings, and it is actually a very complicated process. Think about the file system tree, speculate about the high-level steps, and try to identify situations when a path lookup might fail.)
- What does it mean to mount a file system?
- What happens when you mount a file system on top of an existing subtree (what contents do you see when you navigate to the root of that subtree, i.e. the mount-point)?

**fsync**

- What is the outcome of calling `fsync` (i.e., can you describe possible initial and final states of a file before/after calling `fsync`)?
- Why do we need `fsync` at all? Why not immediately persist all data?
- Based on this, what guarantees does the `write` system call provide?
- *Thinking forward to next class*: Given the granularity of access supported by block devices, what types of things can go wrong when an application calls `fsync` if proper care is not taken in the application/file system implementations? What types of guarantees might be desirable for a system to support? Why are they not standard guarantees?

**rename**

Renaming a file is a seemingly simple task. Yet the deeper you dive into the `rename` system call, the more interesting it becomes. `rename` is the first time we will discuss the concept of atomicity.

- What is the outcome of a successful `rename`?
- What are the ways that `rename` can fail?
  - What errors does it give, and what are the possible states that can result in the system?
- Given this, `rename` is often used to update files. What combination of system calls could you use to perform a series of file modifications so that either all of your modifications are reflected in the final state of the file, or none of them?

**lseek**

The `lseek` system call updates a file data structure's internal offset. This is useful when making non-sequential reads and writes.

- Does `lseek` modify any persistent file state?
- What is the relationship between `lseek`, `read`, `pread`, `write`, and `pwrite`?