CS 333 :: Meeting 01 :: File System API

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 the storage subsystem. 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 key 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?

- inode number
- path
- file descriptor

Questions

- Which of the three identifiers must be unique and in what namespaces/contexts are they unique (per-process, component-wise, and/or globally unique)?
- Which identifiers are easy to remember? Have intuitive interpretations?
- What types of relationships do each identifier type allow us to specify?
- What are the relationships between each type of identifier and the two types of links: **hard** and **symbolic**?
- Can you think of any other types of identifiers that could be used to describe a logical unit of storage (e.g., a file, an object)

files (the data structure)

Each process contains a private table that maps *file descriptors* (per-process integer identifiers) to *file* data structures. We should try to be precise when we use the term file: colloquially file 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 always being precise is difficult, so context should be helpful when determining what is meant by the word *file*.

- What field(s) are stored in the file *data structure*?
- What system calls alter those fields, and in what ways?
    - Think of a task you would want to perform on a file. You should be able to describe how the file data structure's state would change after successful completion of that task, and be able to describe the system call(s) that you would execute to produce those changes.
- What are the relationships among 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? The file system's job is to impose structure over an unordered array of blocks; first we will study the block interface and block device characteristics, then how file systems handle these peculiarities.

directories

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

- All directories contain two files by default: `.` and `..`. What are these files and what is their purpose?
  - As a result of these special files, what is the "link count" of an empty directory?
- `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 (i.e., inode number, pathname, or file descriptor, as described 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` ?
- What is a **capability**, and why does the book claim that a file descriptor is a capability?

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 a symbolic link (hint: type `which python` at the command line, and then check the long listing (`ls -l`) of the resulting pathname)

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)?
- What happens when you unmount a file system?

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 the existence of `fsync`, what guarantees does the `write` system call actually provide?
- *Thinking forward to next class*: Given the granularity of access supported by block devices, what types of things could go wrong if an application calls `fsync` and 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 encounter the concept of *atomicity*.

- What is the outcome of a successful `rename`?
- What are the ways that `rename` can fail?
  - What errors does rename report, and what are the possible states that can result in the system when there is a `rename` failure?
- Given the strict requirements of `rename` atomicity, `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 your modifications are reflected in the final state of the file*?

lseek

The `lseek` system call updates a `file` data structure's internal offset. This is useful for issuing non-sequential reads and writes (commonly referred to as random reads and writes, even when the operations are not random in the mathematical sense).

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