

CS 4414/5416: Systems Programming

File Systems

David Bindel

2026-09-29

```
fn horner(x: f64, c: &[f64]) -> f64 {  
    let mut result = 0.0;  
    for ci in c {  
        result = result * x + ci;  
    }  
    result  
}  
  
fn hornerv(xs: &[f64], c: &[f64]) -> Vec<f64> {  
    xs.iter().map(|x| horner(*x, c)).collect()  
}
```

How might you make this faster when `xs` is large?

Logistics

- P2 (fast edit) due tomorrow (09/30)
- P3 (minishell) released
- Keep an eye on Ed (may have an issue Thu)
- Lab 6 is Friday: memory-mapped I/O

- Yes, you can now “do the lab” without coming in
 - But this is also a time/place to work with peers
- Yes, Claude can probably do this for you
- Lab material is fair game for the final

Poll Everywhere!

```
unsafe extern "C" {  
    fn memmove(dst: *mut c_void, src: *const c_void,  
               len: size_t) -> *mut c_void;  
}  
  
fn my_memmove(dst: &mut [u8], src: &[u8]) -> *const u8 {  
    if src.len() > dst.len() { panic!(); }  
    unsafe {  
        memmove(dst.as_mut_ptr() as *mut c_void,  
                src.as_ptr() as *const c_void, src.len()) as *const u8  
    }  
}
```

This code compiles; should I still worry? (5 min, then poll)

- (a) Yes, a safe wrapper cannot return a raw pointer
- (b) Yes, this violates Rust borrow safety rules
- (c) No, this should function as advertised

This code compiles; should I still worry? (5 min, then poll)

- (a) Yes, a safe wrapper cannot return a raw pointer
- (b) Yes, this violates Rust borrow safety rules
- (c) No, this should function as advertised

Recap

This module: Unix abstractions

- Basic abstractions (files, pipes, virtual memory)
- Processes and inter-process communication
- Introduction to TCP sockets

(Yes, some overlap with 3410/4410.)

- Today: file systems, focusing on
 - Persistence
 - Naming
 - Data structure interfaces
- Thu: virtual memory
- Fri: lab on memory-mapped I/O

Unix Filesystem Interface

What's In the Box?

```
(base) dbindel@dhcp-vl2041-13487 lec % ls /
Applications      etc                private           Users
bin               home              sbin              usr
cores             Library           System            var
dev               opt               tmp               Volumes
```

- How does `ls` know what to show?
- What do the entries in `dev` mean?
- What other stuff is hiding here?

I'll take operating systems for 100!

A: Unix

Q: What if everything was a file?

Two key pieces of OS functionality:

- Persistent storage data structures
- A naming system - files, processes, pipes, sockets, ...

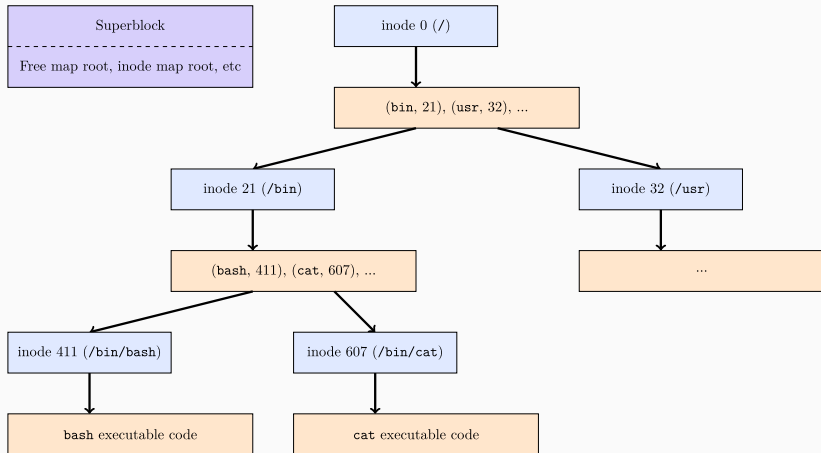
Plus programmatic interfaces to this functionality

Unix FS Data Structures

Abstraction from HW interfaces (e.g. SCSI)

- Storage is a giant logical array (like memory)
- Allow block read/write (usually 512 bytes)
- Access times may be non-uniform (like memory)
- No notion of files, directories, permissions, etc

Persistent Data Structure



(Partial picture; orange boxes represent their own structures)

Components include:

- Tracking of filesystem metadata
- Free list (and maybe inode list)
- Index nodes (inodes) with file info
- Data blocks referenced by inodes
 - May be arranged in tree (or list for FAT)

AKA inodes, they contain metadata:

- File type
- File size
- Ownership and permissions
- Timestamps
- Reference count
- Data block pointers
- etc

Ordinary files are files!

- Just a bag of bytes!
- No text vs binary distinction

Data blocks list (name, inode) pairs

- Always includes
 - (., current dir inode id)
 - (.., parent dir inode id)
 - At root, .. maps to current dir inode
- *Name* vs path (typically at most 255 char)

Does *not* directly include directory path.

Everything is a file! With different inode types

- Devices (disks, keyboards, mouses, NICs)
- Named pipes (FIFOs)
- Unix domain sockets

Unifies naming and interfaces, but no data blocks.

- Standard UNIX access metadata specified
 - File owner and group
 - Read/write/execute permissions for owner, group, world
 - Permissions for owner/group/world in three octal digits
 - Three bits per digit, high to low: read, write, execute
 - Ex: `chmod 700 foo` means owner (but not group or world) can read (4), write(2), execute (1)
- Many systems have richer access control list metadata

Link a file for access via a different path

- *Hard* link (`ln foo bar`)
 - Add directory entry (bar, inode id for foo)
 - Increments the inode reference count
- *Soft* link (`ln -s foo bar`)
 - Create a symbolic link inode referencing foo
 - Add directory entry (bar, inode for bar)

Use `rm` to *unlink* a file

- Removes file from a directory
- Decrements the inode reference count
- If reference count hits zero, free space

Which of the following do you think is true? (5 min, then poll)

- (a) Hard links to directories are not allowed
- (b) Soft links to directories are not allowed
- (c) Linking directories in general is not allowed
- (d) Why would you disallow any linking?

Often *virtualize* file systems (VFS)

- Hook an FS onto an existing FS (mounting)
- Unifies multiple underlying file systems
- Data centers around *vnode* (virtual node)
 - May or may not map to inode on persistent store

Involve in-memory (vs persistent) structure.

Ask Me a Question!

Unix FS Interfaces

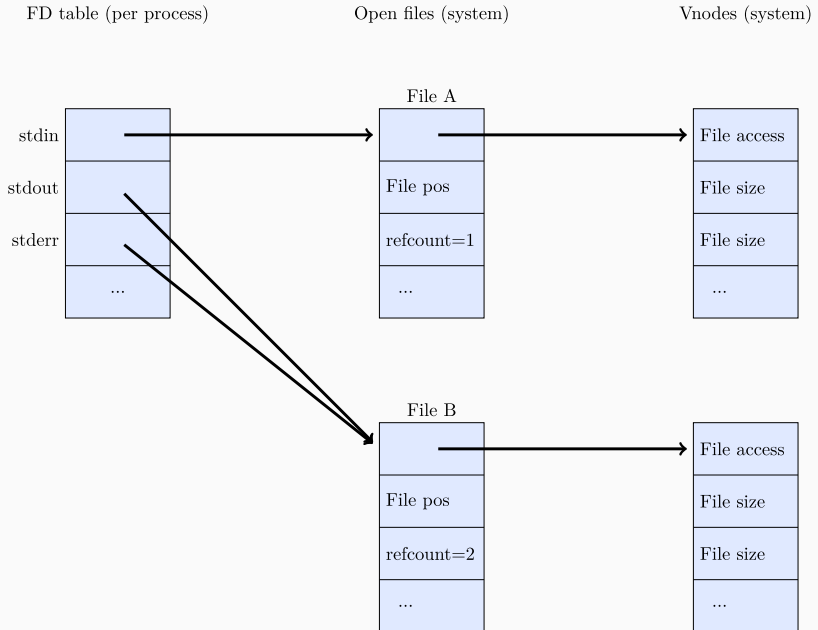
A Simple Example

```
./foo < A.txt > B.txt 2>&1
```

- Standard input from **A.txt**
- Standard output to **B.txt**
- Standard error duplicates standard output

What does the kernel make of this?

What the Kernel Sees



Kernel maintains three-level structure

- Per process file descriptor table
- Open file table (file descriptions)
- Vnode table (VFS analogue of inodes)

Let's consider each

Table of file descriptor pointers

- 0: Standard input
- 1: Standard output
- 2: Standard error

and others via system **open**

Contains file descriptor structs with

- Pointer to vnode
- Reference count
- Offset

This is a *system-wide* data structure

Contains vnodes (virtual nodes)

- Metadata about virtual files
- Copy of inode for regular files (roughly)
- May have vnodes with no associated inode!

This is again a *system-wide* data structure

Why Not Per-Process Positions?

Matters for sharing! Consider

```
( echo "-- Makefile --" ; cat Makefile ) > foo.txt
```

- Open paren starts a new shell process
- Redirect sets `stdout` to `foo.txt`
- `echo` is a built-in, advances output position
- `cat` is a new subprocess!
 - It inherits parent file descriptors
 - Gets a consistent view of position

Why Three Layers?

- Multiple FD entries to the same open file entry
 - Things like `stderr` and `stdout` the same
 - Or subprocesses sharing `stdout` with parent
 - Latter means it can't be per process
- Multiple open files for same vnode
 - Unrelated readers/writers might not share file pos!

- Default Rust I/O is *unbuffered*
- Can use **BufReader** and **BufWriter**
- Remember buffering when reasoning about position!
 - Kernel read position may be ahead user view
 - Kernel write position may be behind user view
- Important to **flush** buffered writers!
 - Errors during flush in **drop** are ignored

Pipes and Redirection

```
grep "^## " m3-01-fs.qmd | wc -l
```

Can set standard input, output, error

- Redirection: Set to input/output file
- Or set it to a *pipe* connecting processes
 - (Usu) associated with a FIFO **vnode**
 - Not associated with an **inode**...
 - Except for named pipes (next week)

Wrap-Up

Unix filesystem gives

- Persistence
 - Via data structures on drive or SSD
- Consistent naming
 - Of files, directories, pipes and sockets, etc
- Standardized interface

Plus three-layer structure to manage shared access

- This Thu: Virtual memory
- Next week: Inter-process communication

- P2 is due tomorrow at 11:59 PM
- P3 is out (writing a mini shell)
- Lab 5 is Friday (mmap and I/O)

On Ed: What was the muddiest point today?