

CS 4414/5416: Systems Programming

Inter-Process Communication

David Bindel

2026-10-08

Logistics

- P3 (minishell) due Monday, 10/19
- Lab 7 is Friday: building pipelines
- No class 10/19 - Fall Break!

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.)

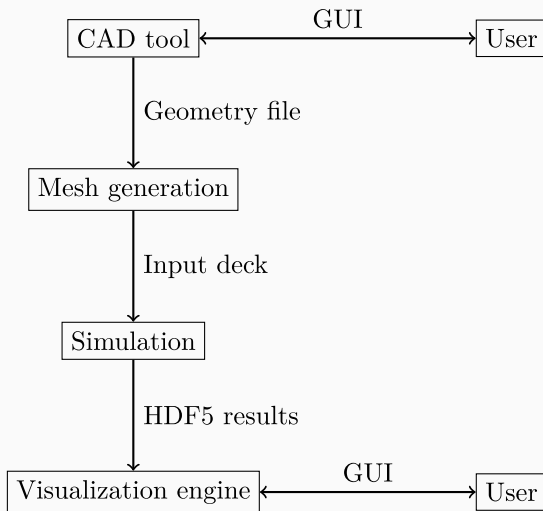
- Tue: processes and IPC intro
- Today:
 - Case studies in IPC
 - Using anonymous and named pipes
 - Using Unix domain sockets
 - A hint of shared memory IPC
- Fri: lab on Unix pipelines

Case Studies

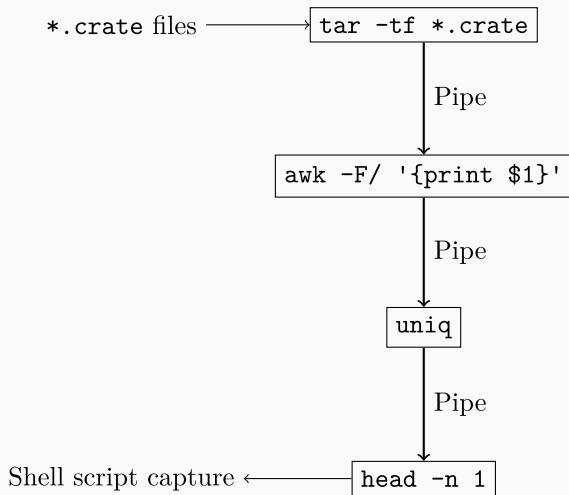
Many Possible Architectures!

Different ways to organize; for each, let's talk:

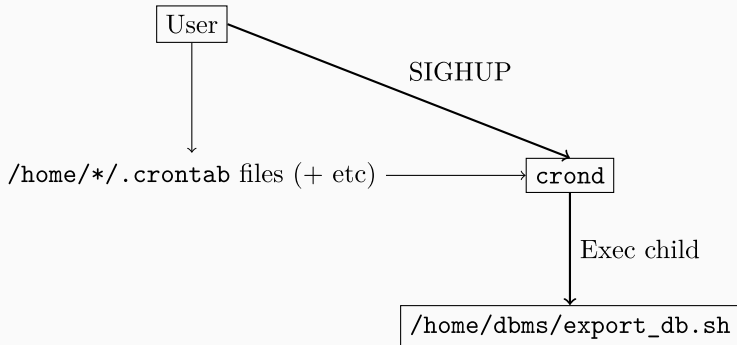
- What are the advantages to this approach?
- What are potential disadvantages and limits?

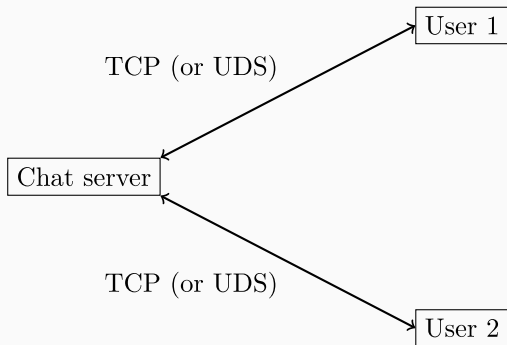


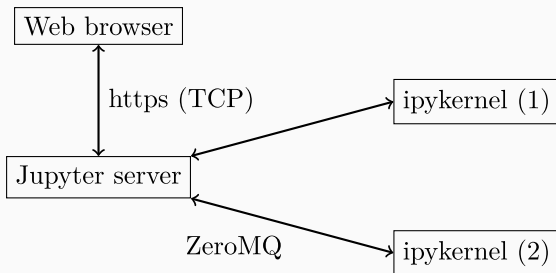
Unix Shell Pipelines



Cron Daemon (System V)







- Jupyter Spawner launches kernels (not shown)
- ZeroMQ runs over several transport layers

Pipes and FIFOs

Pipe properties:

- Unidirectional: writer output to reader input
 - Bidirectional pipes are supported a few places
- Ordered (FIFO), streaming data (vs messages)
- Afford *limited* internal buffer capacity

- `mkfifo` creates a named pipe (FIFO) inode
- Anonymous pipes may involve vnode, but not inode
- Data still managed by kernel ring buffer
 - Data doesn't go through disk!

Communication via Named Pipes

```
rm -rf cmd-pipe result-pipe # Clean up
mkfifo cmd-pipe             # Create pipe for input commands
mkfifo result-pipe          # Create pipe for results

./server < cmd-pipe > result-pipe
./client > cmd-pipe < result-pipe
```

- Can connect named pipes from the command line
- Can use with things like `cat`
- Great for debugging!

Common design case:

- Front-end GUI or CLI for user interaction
- Back-end engine handles heavy compute
- Defined protocol for connecting the two
 - Allows mix-and-match: multiple GUIs and engines

One implementation: two processes talk over pipes

Example: Stockfish

- Stockfish is a strong open-source chess engine
- Interaction via Universal Chess Interface (UCI)
- En Croissant and other GUIs spawn Stockfish with communication pipes

- How do we maintain consistent distributed state?
- How do we track software version inconsistencies?
- How do we handle partial failure?
- How do we debug issues?
- What's with all the buffers?

Let's talk through this with Stockfish!

- Both GUI and engine maintain play state separately
- Command `ucinewgame` to start a new game
- Command `position` can make move *or* re-describe board
- Stockfish checks move validity just in case!

- The `uci` command is run at the outset
- Engine responds with `id` and `option` commands to GUI
- Engine concludes with `uciok`
- GUI will kill engine if no `uciok` by timeout

- GUI can detect child termination with signals
- What about hang vs long computation?
 - `isready` / `readyok` to check if ready for commands
 - `stop` to ask engine to stop current thinking ASAP

- Can tell Stockfish to keep a log!
- Can also run Stockfish from command line interface
 - UCI is a text protocol!

Stockfish is C++ vs Rust, but:

- Uses `std::endl` to flush buffers when appropriate
- Uses specialized synchronized versions of standard streams
 - This is mostly about thread safety
 - Have special `sync_cout` and `sync_endl` for this

Lab 7 will use named pipes with

- Server that registers clients and sends broadcasts
- Client can be just `cat < client-pipe`

Ask Me a Question!

Unix Domain Sockets

Socket is a thing you connect to

- Don't scream into the void – talk to someone!
- Can have multiple simultaneous connections
 - This is unlike pipes
- Once connected, can communicate

- Stream as phone call
 - Reliable, ordered stream
 - Similar to style of paired FIFOs
- Datagram as mail
 - Unreliable, unordered messages

On the server side (for Unix or TCP sockets):

- *Create* the socket
- *Bind* socket to a name
 - Hard for clients to connect otherwise!
 - Make sure name is not already used
- *Listen* for connection requests
- *Accept* to get new descriptor for a connection

```
use std::os::unix::net::UnixListener;
use std::io::Write;

fn main() -> std::io::Result<()> {
    let args: Vec<String> = std::env::args().collect();
    let listener = UnixListener::bind(&args[1])?;
    let mut ticket = 0;
    for stream in listener.incoming() {
        let Ok(mut stream) = stream else { continue; };
        writeln!(stream, "You get ticket #{ticket}")?;
        ticket += 1;
    }
    Ok(())
}
```

This is a *stream* socket

- Bidirectional - available for read and write
- Reliable - won't silently lose data
- Ordered

Use `std::os::unix::net::UnixDatagram` for datagrams

Client Connections

```
use std::os::unix::net::UnixStream;
use std::io::Read;

fn main() -> std::io::Result<()> {
    let args: Vec<String> = std::env::args().collect();
    let mut stream = UnixStream::connect(&args[1])?;
    let mut response = String::new();
    stream.read_to_string(&mut response)?;
    println!("{response}");
    Ok(())
}
```

- `UnixStreams` are closed on `drop`
 - Close blocks until all data transmitted
- Can also call `shutdown` (immediate return)
 - Takes argument: `Read`, `Write`, or `Both`
- All also true for TCP streams

A lot like a Unix socket!

- But different library interfaces
- Names in a pseudo-filesystem (no inode)
- Unidirectional or bidirectional (half-duplex or full-duplex)
- Can go over network (Server Message Block protocol)

Which is true of both named pipes and Unix sockets?

- (a) Communication is bidirectional
- (b) Naming goes through the file system
- (c) A single pipe/socket can be used by multiple clients to talk to a server simultaneously
- (d) Communication can be datagram- or stream-oriented

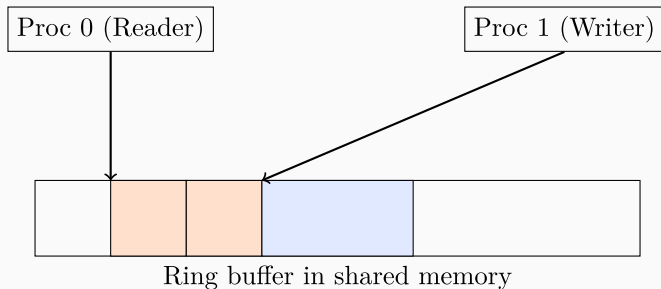
Shared Memory

- Shared memory can have zero copies through buffers!
- Can be *much* more efficient for large messages

It's a tremendous pain to get right! Manage

- Protocol to set up shared pages
- Protocol for synchronizing reads/writes
- Proper alignment on allocations

- Shmipc: high-perf shared memory IPC library (ByteDance)
- Implementations in several languages (Rust, Go, C++)
- System negotiates shared page setup
- Shared ring buffer structure with aligned allocations
 - Hardware-level ordering (lock-free with atomics)
- Rust version uses memmap2



- Writer requests allocation from ring
- Reader accesses parts no longer written
- Atomic/sequentially consistent pointer updates

Wrap-Up

- IPC enables a variety of architectures
- Several low-level mechanisms: pipes, sockets, shm
• Can wrap these with higher-level abstractions

- Fall Break!
 - No lecture Tue, 10/13
 - Adrian Sampson on 10/15 - intro TCP sockets
 - No Bindel OH next week
- P3 final due 10/19
- Lab 7 tomorrow: using Unix pipes

Enjoy your break!