little languages

lecture 31:

Mystery

We will be on the VM today. Go ahead and pull.

Also be thinking about and discussing:
What do you want to spend the last few lectures discussing?

What do you want to spend the last few lectures discussing?

- Discuss with your neighbors for 2 minutes!
- Add to the list on PollEv and/or vote for other people's.

`tail` Utility Program

- The **tail** utility prints the last 10 lines of a file by default
- The **-n** or **--lines=N** flag allows you to change the number of lines
- The **-f** or **--follow** flag will output any data appended to a file as it grows
- The follow flag is useful for "watching" files output by programs
  - For example, if you have a program that is logging data to a file (common in web applications) you an "tail" the log file to see its output in real time.

```rust
use std::fs::File;
use std::io::prelude::*;

fn main()  -> std::io::Result<()> {

    let mut file = File::create("foe.txt")?;
    let mut i = 0u64;

    loop {
        i += 1;
        file.write_all(format!("file  : {}\n", i).as_bytes())?;
        println!("stdout: {}", i);
        eprintln!("stderr: {}", i);
    }

}
```

Mystery on the Pipeline Express

- Open two terminals and **ssh** into both of them.
- Organize them side-by-side.

- Navigate to the 590-material/lec31-pipes directory.

- First, in the left hand terminal, run the following command:
  - **cargo run | less**

- Second, in the right hand terminal, run the following command:
  - **tail -f foe.txt**

- Then, in the left hand terminal, press the space bar slowly a few times until you see some changes on the right-hand side.

1. Diagram how foe and less are related via stderr/stdout/stdin.
2. Try to explain the behavior based on what you observe.
3. Press q in less. Try to explain the error. Feel free to start **cargo run | less** up again

Part 7. Consider the following Rust program named `foe`. It creates a blank new file named "foe.txt". It then infinitely loops. In each loop iteration it writes an 'o' to `stdout`, *then* writes an 'e' to `stderr`, and *then* writes an 'f' to "foe.txt".

```
use std::fs::File;
use std::io::prelude::*;

fn main()  -> std::io::Result<()> {
    let mut file = File::create("foe.txt")?;
    loop {
        println!("o");
        eprintln!("e");
        file.write_all(format!("f\n").as_bytes())?;
    }
}
```

There is a command-line utility program called `sleep`. It is a program that simply sleeps (pauses) for a specific amount of time (in seconds) and then exits. *It does not read any input nor produce any output.* Usage: `sleep <seconds>`.

So you try running the command: `foe | sleep 5` and you see a blur of *only e's* printed to your screen *very briefly*. The e's stop printing in under 1 second. After 5 seconds, you see the following Rust panic error message being printed by `foe`: "failed printing to stdout: Broken pipe", the command exits, and you're back at a command prompt. You check the file size of "foe.txt" and it is 65,536 bytes which is exactly 2^{16} bytes.

7.1. Draw a diagram of how your terminal and the processes for `foe` and `sleep` are connected in terms of `stdin`, `stdout`, and `stderr`. Draw a `pipe` wherever one is connecting an output stream to an input stream (representing a `pipe` with a rectangle is fine).

The questions on this page refer to the scenario from the previous page.

7.2. When you run `foe | sleep 5`, why are *only e's* printed to your terminal's screen?

7.3. Given `sleep` waits 5 seconds to do nothing (it reads no input from `stdin` and writes no output) and exit, what can you infer the "broken pipe" error means in this scenario?

(Part 7 Continued) There is a another command-line utility program called `timeout` that will run a command for the shorter of the command exiting or a time limit in seconds.

Its signature is: `timeout <seconds> <command>`

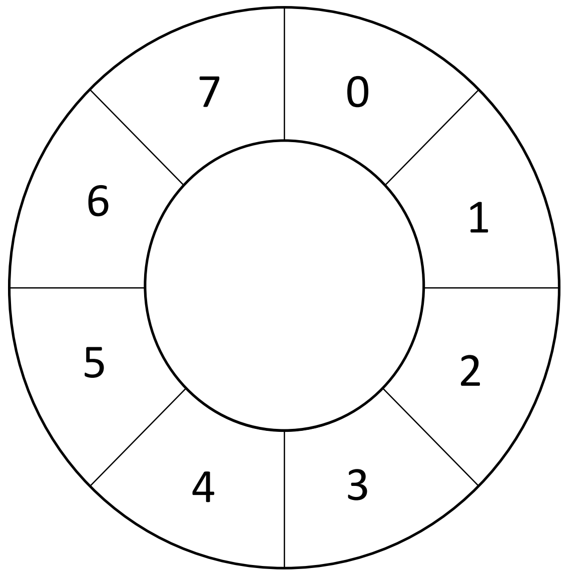
When you run `timeout 5 foe` you see a blur of alternating o's and e's printed to your screen. Immediately after the 5 seconds is up the printing stops and you're right back at a command prompt. You check the file size of "foe.txt" and see that the first time you try this it's 812,814 bytes. The second time the file is 915,584 bytes. The third 813,764.

7.4. Given these two scenarios, how much data (in terms of bytes) can you infer a pipe is able to store before it becomes full?

7.5. From these scenarios, what can you infer happens to any process that is writing to a pipe that is full? (Hint: reread the output of both scenarios.) What evidence led you to that conclusion?

EC. What process must manage the memory of a pipe: `foe`'s, `sleep`'s, or an operating system process?

Circular Buffers



- A Pipe is a circular buffer connecting the output of one process to the input of another.
    - A circular buffer is really just a clever use of a fixed sized array!
- Useful when two processes or threads need to stream data to each other
- The writing side is called the producer, the reading side is called consumer
    - Each maintains the index up to which they've written or read respectively
    - The reader can never progress past the writer (the reader waits until writer progresses)
- When the writer reaches the end of the buffer, it *circles back* with
    - Clever, simple arithmetic: writerIndex % bufferLength
    - The writer can never progress past the reader (the writer waits until reader progresses)
- As long as both sides are making progress, allows for a continuous stream of data to be efficiently transferred within a fixed block of memory.
- Visualization: https://en.wikipedia.org/wiki/Circular_buffer

Part 5. For reference, the definition of `NFA`, `State`, and `Char` are provided below:

```
pub struct NFA {
    start: StateId,
    states: Vec<State>,
}

enum State {
    Start(Option<StateId>),
    Match(Char, Option<StateId>),
    Split(Option<StateId>, Option<StateId>),
    End,
}

enum Char {
    Literal(char),
    Any,
}
```

Suppose you want to *generate* a random input string acceptable by an `NFA` in a method defined in the `NFA`'s `impl` block with the following signature:

```
impl NFA {
    pub fn gen(&self) -> String { /*...*/ }
}
```

5.1. How would you implement such a method in linear time with respect to the length of the generated string? Respond with English, pseudo-code, and helper function descriptions as necessary. *Be sure your response addresses how all variants of `State` and `Char` would impact your algorithm.* You can assume `randomBool` and `randomChar` functions exist.

Suppose you want to overide the addition operator for `NFA`, whose definition is on the previous page, such that adding two `NFA` structs together *concatenates* the NFAs. For example:

```rust
let ab: NFA = NFA::from("ab");
let cd: NFA = NFA::from("cd");
let abcd: NFA = ab + cd;
if abcd.accepts("abcd") {
    println!("This should _always_ print!");
}
```

The `impl` of the `Add` trait for overloading this operator is as follows:

```rust
impl Add for NFA {
    type Output = NFA;

    fn add(self, rhs: NFA) -> NFA { /* ... */ }
}
```

5.1. Given the `State` values are stored in an *Arena* and use `StateId` values to refer to one another, what is the *primary challenge* of forming a bigger `NFA` out of two smaller `NFA`s?

5.2. How would you implement the `add` method? Use English, psuedo-code, and helpers as necessary. Notice neither `self` nor `rhs` are `mut`.

Final Problem Set - Vote

- Option A – Extend \`thegrep\` with true implementations of exam questions:
  - Generating random strings that match
  - Operator overloading (at least + for catenation, but probably also | for alternation)
  - Perhaps the addition of + or ? regex operators
  - Release: Tonight – Due LDOC

- Option B – Implement \`thmake\` - mini make problem set
  - Implement a directed acyclic graph
  - Implement topological sort algorithm
  - Read file modification times and develop plan to execute recipes
  - Release: Weds Night – Due LDOC