The Shell & Processes (fin.)

Intermediate Computer Systems Programming, Fall 2025

Instructor: Travis McGaha

TAs: Theodor Bulacovschi Ash Fujiyama

pollev.com/tqm

- How are you? Any feedback?

Administrivia

- HW06 posted after class today
  - Not everything you need will be talked about till Wednesday
  - Will have extended deadline due to fall break
- Check-in 06 posted after class
  - Just finishing something you may not finish in class today
  - Keeping it due by end of day Monday so that we can process re-opens in a timely manner
- Midterm Details
  - In-class on Wed Oct 22nd
  - Posted soon

Lecture Outline

- **The Shell**
- stdin, stdout, & redirection
- pipe()
- Processes in other languages

Unix Shell

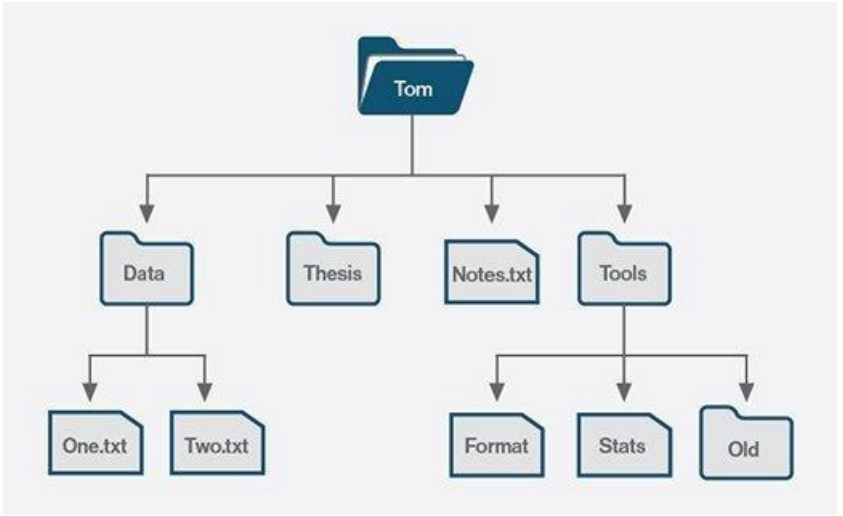
- A **user level** process that reads in commands
  - This is the terminal you use to compile, and run your code
- Commands can either specify one of our programs to run or specify one of the already installed programs
  - Other programs can be installed easily.
- There are many different shells, in this class we use ***Bash***
  - Others like zsh, fish, etc exit.
- There are many commonly used bash programs, we will go over a few and other important bash things.

Current Working Directory & Hierarchical File System

- Folder and Directory are pretty much synonyms. Technically there is a difference, but it is not worth covering.

- In some ways a shell is like File Explorer or Finder
  - Has a concept of a "***Current Working Directory***" which is the directory we are in right now
  - We change which directory we are in and can use it to explore the contents of other directories as we wish.

- Directories can contain other Directories
  - **Subdirectory** is used to describe a directory contained in another
  - a few directories being the "overall root"
  - "parent" and "child" terminology returns here.



./..

- "/" is used to connect directory and file names together to create a file path.
  - E.g. "**`workspace/595/hello/`**"

- "." is used to specify the current directory.
  - E.g. "**`./test_suite`**" tells to look in the current directory for a file called "**`test_suite`**"

- ".." is like "." but refers to the parent directory.
  - E.g. "**`./example/../test_suite`**" would be effectively the same as the previous example.

UNIX Design Philosophy

- Philosophy behind development of UNIX that spread to standards for developing software generally.
  - Arguable more influential than UNIX itself

- Short version:
  - Programs should "Do One Thing And Do It Well."
  - Programs should be written to work together
  - **Write programs that handle text streams, since text streams is a universal* interface.**

Unix Shell Commands

- Commands can also specify flags
  - E.g. "`ls -l`" lists the files in the specified directory in a more verbose format

- Revisiting the design philosophy:
  - Programs should "Do One Thing And Do It Well."
  - Programs should be written to work together
  - Write programs that handle text streams, since text streams is a universal interface.

- <u>These programs can be easily combined with UNIX Shell operators to solve more interesting problems</u>

Common Commands (Pt. 1)

Going over these quick,
these are here for your reference.

- "**`ls`**" lists out the entries in the specified directory (or current directory if another directory is not specified
- "**`cd`**" changes directory to the specified directory
  - E.g. "**`cd ./solution_binaries`**"
- "**`exit`**" closes the terminal
- "**`mkdir`**" creates a directory of specified name
- "**`touch`**" creates a specified file. If the file already exists, it just updates the file's time stamp

Common Commands (Pt. 2)

Going over these quick,
these are here for your reference.

- "**`echo`**" takes in command line args and simply prints those args to stdout
  - "**`echo hello!`**" simply prints "**`hello!`**"
- "**`wc`**" reads a file or from stdin some contents. Prints out the line count, word count, and byte count
- "**`cat`**" prints out the contents of a specified file to stdout. If no file is specified, prints out what is read from stdin
- "**`head`**" print the first 10 line of specified file or stdin to stdout

Common Commands (Pt. 3)

Going over these quick,
these are here for your reference.

- "**`grep`**" given a pattern (regular expression) searches for all occurrences of such a pattern. Can search a file, search a directory recursively or stdin. Results printed to stdout
- "**`history`**" prints out the history of commands used by you on the terminal
- "**`cron`**" a program that regularly checks for and runs any commands that are scheduled via "crontab"
- "**`wget`**" specify a URL, and it will download that file for you

The shell just fork-exec's your commands*

- Whenever you type in a command like `echo hello`
  - `echo` is the name of a program (just like test_suite or cowsay)
  - By default the shell will search in `/bin/` for a program of specified name and fork-exec it
  - `execvp` will automatically search `/bin/` for you

- When we have a `./` before the name (like `./test_suite`) it tells us to look in the current directory instead of `/bin/`

- A shell doesn't implement "`echo`" specifically
  - It just forks a process that execvp's echo.

Ed Discussion

- Lets implement a simple shell!

- See Ed ☺

Lecture Outline

- The Shell
- **stdin, stdout, & redirection**
- pipe()
- Processes in other languages

stdout, stdin, stderr

- By default, there are three “files” open when a program starts
  - stdin: for reading terminal input typed by a user
    - `cin` in C++
    - `System.in` in Java
  - stdout: the normal terminal output.
    - `cout` in C++
    - `System.out` in Java
  - stderr: the terminal output for printing errors
    - `cerr` in C++
    - `System.err` in Java

stdout, stdin, stderr

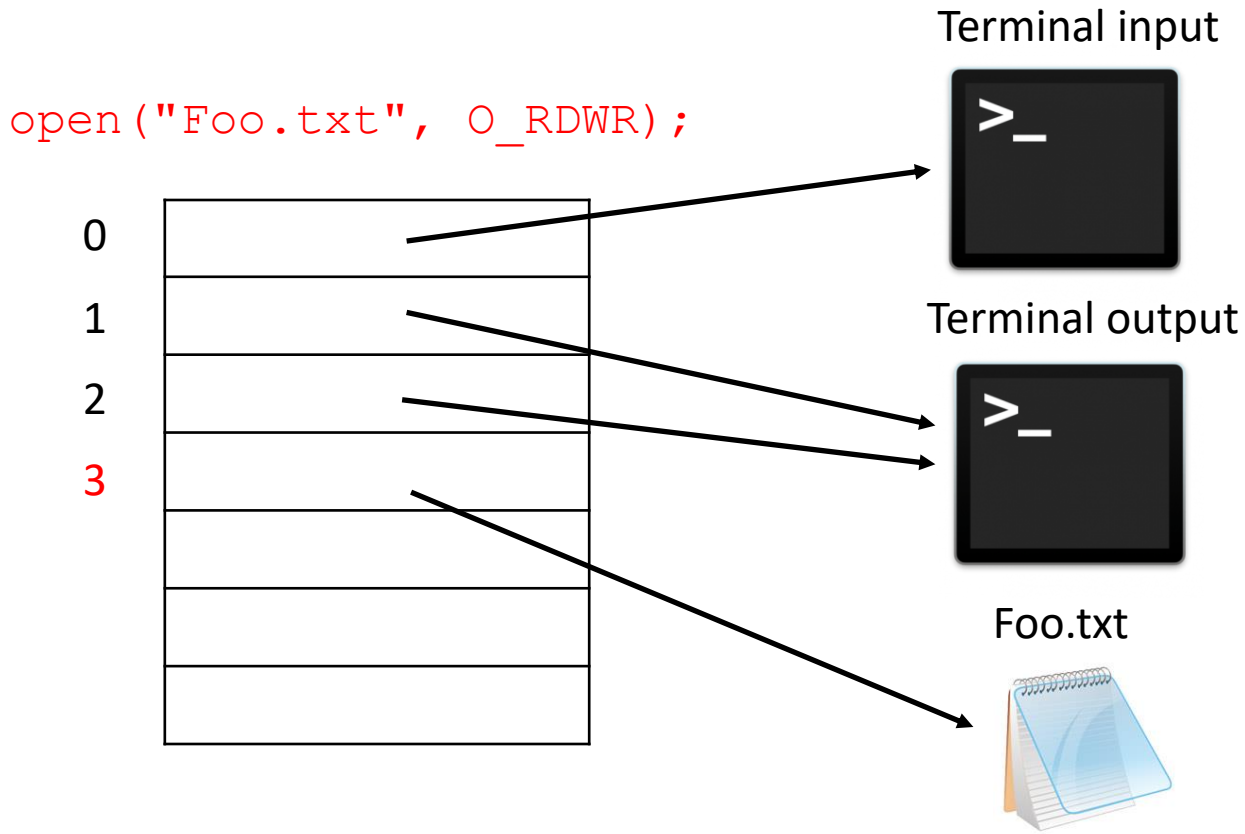
- stdin, stdout, and stderr all have initial file descriptors constants defined in `unistd.h`
  - `STDIN_FILENO  -> 0`
  - `STDOUT_FILENO  -> 1`
  - `STDERR_FILENO  -> 2`

- These will be open on default for a process

- Printing to stdout with cout will use `write(STDOUT_FILENO, …)`

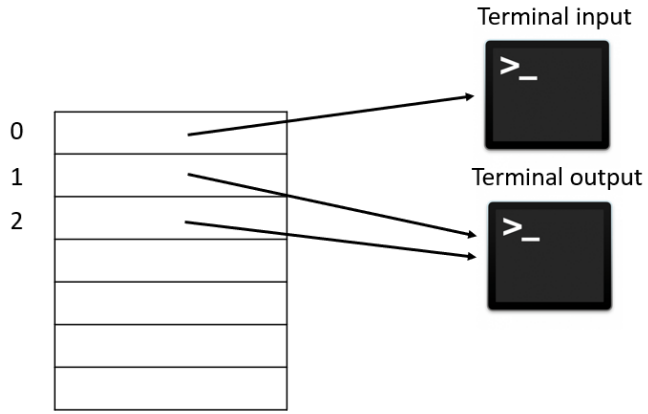
File Descriptor Table

- In addition to an address space, each process will have **its own file descriptor table** managed by the OS
- The table is just an array, and the file descriptor is an index into it.



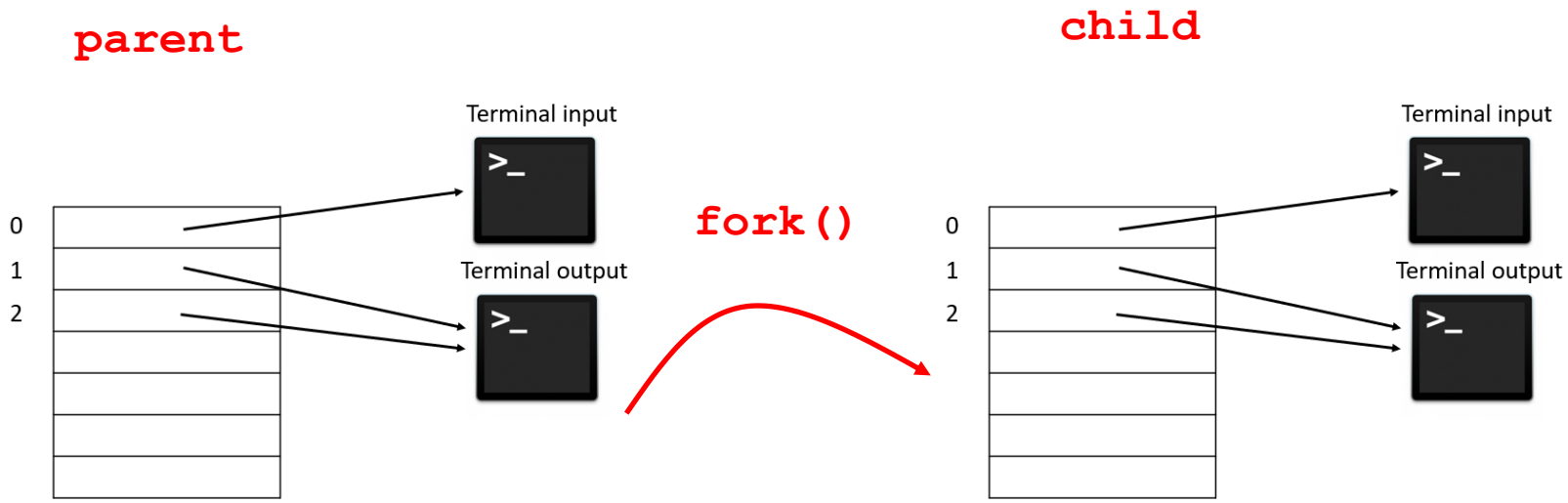
File Descriptor Table: Per Process

- each process will have **its own file descriptor table** managed by the OS
- Fork will make a copy of the parent's file descriptor table for the child



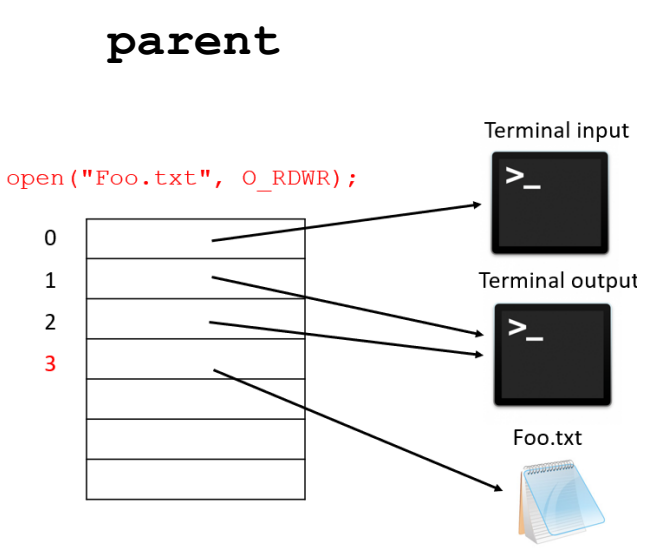
File Descriptor Table: Per Process

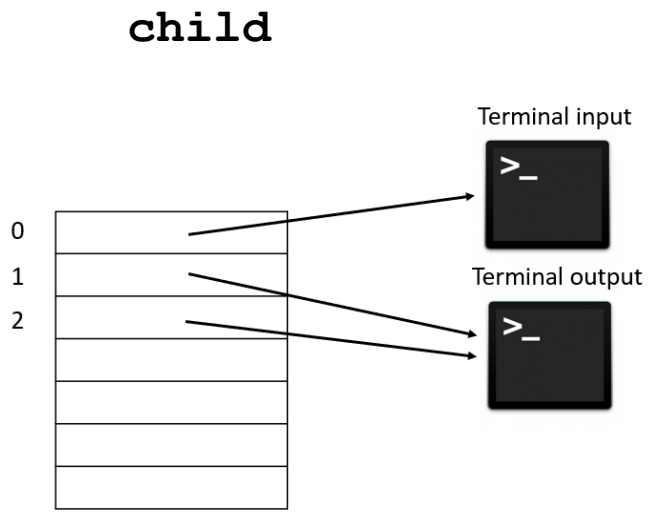
- each process will have **its own file descriptor table** managed by the OS
- Fork will make a copy of the parent's file descriptor table for the child



File Descriptor Table: Per Process

- each process will have **its own file descriptor table** managed by the OS
- Fork will make a copy of the parent's file descriptor table for the child





Child is unaffected by parent calling open!

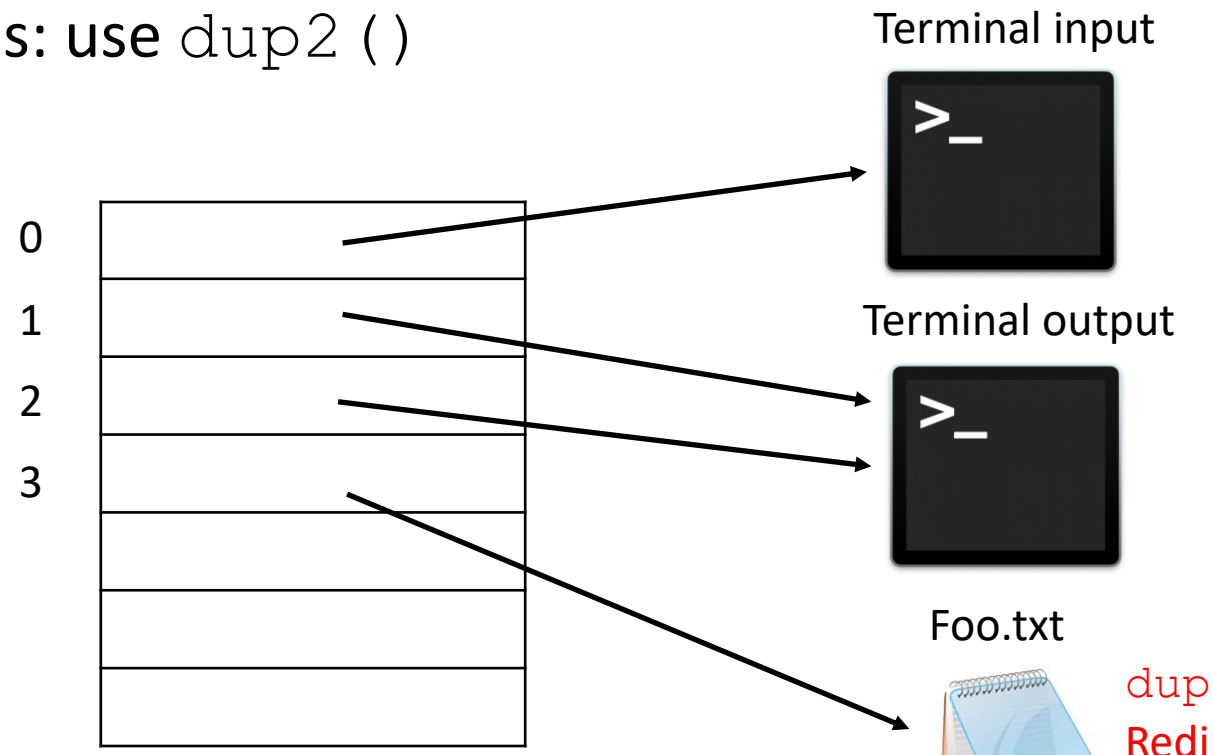
Gap Slide

- Gap slide to distinguish we are moving on to a new example (that looks very similar to the previous one)

Redirecting stdin/out/err

`printf` is implemented using `write(STDOUT_FILENO`
That's why it is redirected
after changing stdout

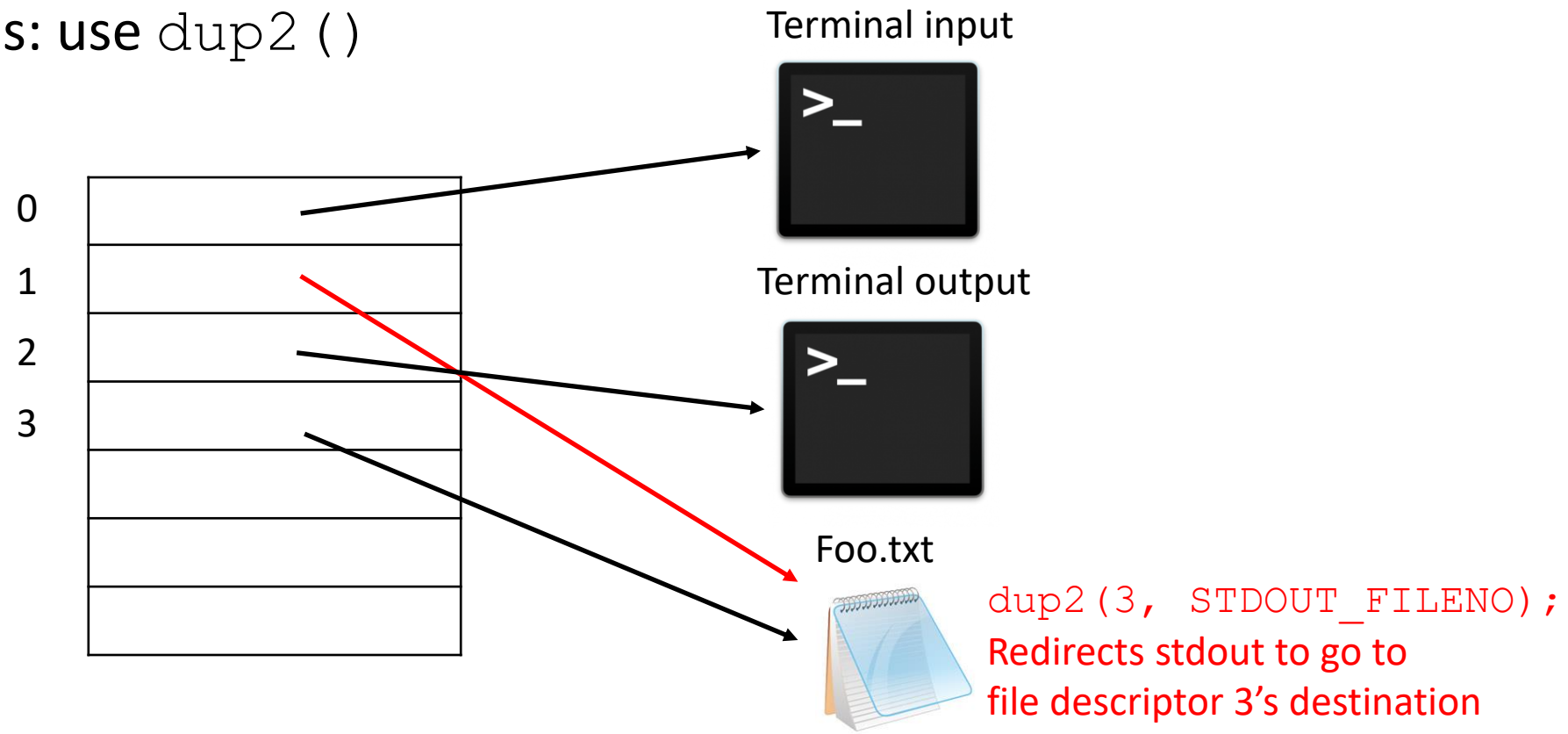
- We can change things so that STDOUT_FILENO is associated with something other than a terminal output.
- Now, any calls to printf, cout, System.out, etc now go to the redirected output
- To do this: use `dup2()`



`dup2(3, STDOUT_FILENO);`
Redirects stdout to go to
file descriptor 3's destination

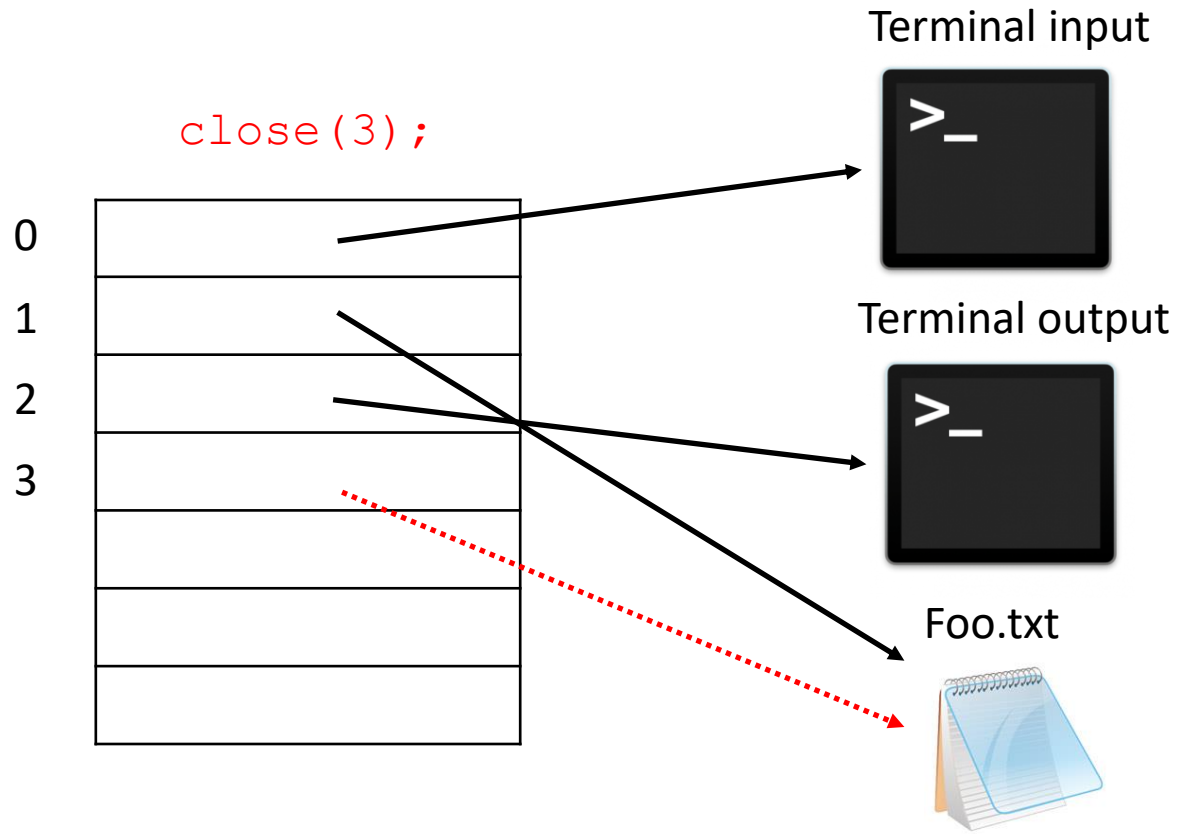
Redirecting stdin/out/err

- We can change things so that STDOUT_FILENO is associated with something other than a terminal output.
- Now, any calls to printf, cout, System.out, etc now go to the redirected output
- To do this: use `dup2()`



Closing a file descriptor

- If we close a file descriptor, it only closes that descriptor, not the file itself
- Other file descriptors to the same file will still be open
- use `close()`



dup2()

- `int dup2(int oldfd, int newfd);`

  File descriptor

  - Creates a copy of the file descriptor **`oldfd`** using **`newfd`** as the new file descriptor number
  - If **`newfd`** was a previously open file, it is silently closed before being reused
  - Returns -1 on error.

Lecture Outline

- The Shell
- stdin, stdout, & redirection
- **pipe()**
- Processes in other languages

Pipes

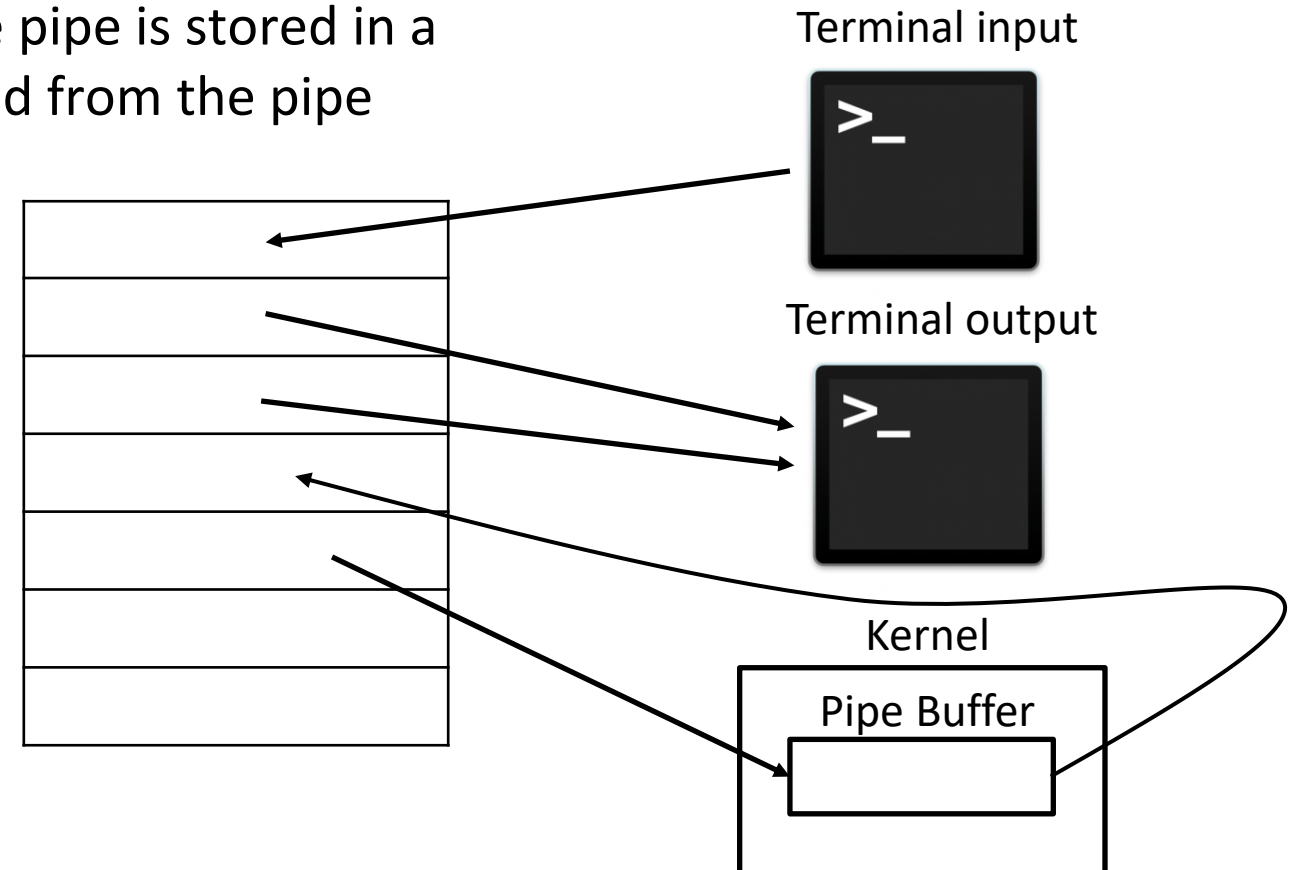
```
int pipe(int pipefd[2]);
```

- Creates a unidirectional data channel for IPC
- Communication through file descriptors! // POSIX ☺
- Takes in an array of two integers, and sets each integer to be a file descriptor corresponding to an "end" of the pipe
- `pipefd[0]` is the reading end of the pipe
- `pipefd[1]` is the writing end of the pipe

- **In addition to copying memory, fork copies the file descriptor table of parent**
- Exec does NOT reset file descriptor table

Pipe Visualization

- A pipe can be thought of as a "file" that has distinct file descriptors for reading and writing. This "file" only exists as long as the pipe exists and is maintained by the OS.
  - Data written to the pipe is stored in a buffer until it is read from the pipe



Pipes & EOF

- Many programs will read from a file until they hit EOF and will not terminate until then
- Like reading from the terminal, just because there is nothing in the pipe, does not mean nothing else will ever come through the pipe.
  - EOF is not read in this case

- EOF is only read from a pipe when:
  - There is nothing in the pipe
  - All write ends of the pipe are closed

- **Good practice: CLOSE ALL PIPE FDS YOU ARE DONE WITH**

Ed Discussion

- Given the following code, what is the contents of "hello.txt" and what is printed to the terminal?

```
using std::flush; // similar to endl. flush buffer but don't print a newline

int main() {
  int fd = open("hello.txt", O_WRONLY);

  cout << "hi" << flush;

  close(STDOUT_FILENO);
  cout << "?" << flush;

  dup2(fd, STDOUT_FILENO);
  cout << "!" << flush;

  close(fd);
  cout << "*" << flush;
}
```

Ed Discussion

```
int main() {
  array<int, 2> pipe_fds;
  pipe(pipe_fds.data());

  pid_t pid = fork();

  if (pid == 0) {
    // child process
    // close the end of the pipe that isn't used
    close(pipe_fds.at(0));

    dup2(pipe_fds.at(1), STDOUT_FILENO);
    string greeting {"Hello!"};

    cout << greeting << flush;

    optional<string> response = wrapped_read(pipe_fds.at(1));
    if (response.has_value()) {
      cout << response.value() << endl;
    }

    exit(EXIT_SUCCESS);
  }
  // continued in the code block to the right
```

- What does the parent print? What does the child print? why? (assume pipe, close and fork succeed)

  pipe_unidirect.cpp
  on course website

```
  // parent
  /// close the end of the pipe I won't use
  close(pipe_fds.at(1));

  optional<string> message = wrapped_read(pipe_fds.at(0));
  if (message.has_value()) {
    cout << message.value() << endl;
  }

  string greeting{"Howdy!"};
  wrapped_write(pipe_fds.at(0), greeting);

  int wstatus;
  waitpid(pid, &wstatus, 0);

  return EXIT_SUCCESS;
}
```

Pipes & EOF

- Many programs will read from a file until they hit EOF and will not terminate until then
- Like reading from the terminal, just because there is nothing in the pipe, does not mean nothing else will ever come through the pipe.
  - EOF is not read in this case

- EOF is only read from a pipe when:
  - There is nothing in the pipe
  - All write ends of the pipe are closed

- This is true for other streams like network communication
- **Good practice: CLOSE ALL PIPE FDS YOU ARE DONE WITH**

Redirecting in the shell

- You can redirect stdin / stdout /stderr for programs in the shell to have them read / write from a file instead of the shell
- `wc < example.txt`
  - wc reads from example.txt instead of the terminal
- `echo hello > some_file.md`
  - Echo hello prints into the specified file instead of terminal
- `cat example.cpp >> some_file.md`
  - Appends the output onto the end of a file
- `make tidy-check 2> err.txt`
  - Capture stderr instead of stdout
- `make tidy-check &> all_output.txt`
  - Captures both stdout and stderr

Pipe in the shell

- Pipes can also be used in the shell to combine commands in the shell

- One of the more useful commands I use:

```
history | grep valgrind
```

  - Runs history but feeds its output as the input to grep.
  - Grep then searches history's output for each line that contains the target and prints it
  - Very useful for looking up commands that you may have forgotten!

Lecture Outline

- The Shell
- stdin, stdout, & redirection
- pipe()
- **Processes in other languages**

Fork-exec

- Fork-exec lets us write programs that do what can be done in the shell
  - We can execute other programs from our program
  - Those other programs can be written in any language! As long as it can run on your system

- This functionality is a fundamental tool.

- This is an Immensely useful tool so it can be found in other languages:
  - Java has the **`RunTime`** class
  - Python has the **`subprocess`** module
  - Rust has the **`Command`** API
  - Node.Js has the **`child_process`** module
  - Usually, it is a bit more user friendly than what we have in C and C++

Example: python subprocess module

```
import subprocess

# this looks really pretty because none of the error handling or parsing is around this

proc = subprocess.run("clang++-15 -o hello hello.cpp", shell=True, stderr=subprocess.PIPE,
                      stdout=subprocess.PIPE, timeout=60)


proc = subprocess.run("./hello", shell=True, stderr=subprocess.PIPE, stdout=subprocess.PIPE,
                      timeout=60)


# get the stdout, stderr, and returncode of the process
stdout = proc.stdout.decode()
stderror = proc.stderr.decode()
returncode = proc.returncode
```

That's all for now!

- Next time:
  - Threads & Concurrency!

- Hopefully you are doing well ☺