OS: Shell & File Descriptors

Computer Systems Programming, Spring 2025

Instructor: Travis McGaha

Teaching Assistants:

Andrew Lukashchuk
Angie Cao
Aniket Ghorpade
Ashwin Alaparthi
Austin Lin
Hassan Rizwan
Lobi Zhao
Pearl Liu
Perrie Quek

Poll Everywhere

pollev.com/tqm

- How are you?

Administrivia

- retry_shell (HW04)
  - Due 2/21
  - Should have everything you need after the first part of this lecture
  - Tests cases & autograder posted tonight or tomorrow (Sorry for delay)
  - Demo later in this lecture.

Lecture Outline

- waitpid() and exit status
- Brief History of Unix & Linux
- Unix Shell & hierarchical file system
- File descriptor System Calls
- File Descriptor Table & Redirections
- Pipe (start)

Processes & Fork Summary

- Processes are instances of programs that:
  - Each have their own independent address space
  - Each process is scheduled by the OS
    - Without using some functions we have not talked about (yet), there is no way to guarantee the order processes are executed
  - Processes are created by fork() system call
    - Only difference between processes is their process id and the return value from fork() each process gets

More: `waitpid()`

```
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

- Calling process waits for a child process (specified by **`pid`**) to exit
  - Also cleans up the child process
- Gets the exit status of child process through output parameter **`wstatus`**
- **`options`** are optional, pass in `0` for default options in *most* cases
- Returns process ID of child who was waited for or `-1` on error

wait() status

- **`status`** output from **`wait()`** can be passed to a macro to see what changed
- **`WIFEXITED()`** true iff the child exited nomrally
- **`WIFSIGNALED()`** true iff the child was signaled to exit
- **`WIFSTOPPED()`** true iff the child stopped
- **`WIFCONTINUED()`** true iff child continued

- Demo: see example in `exit_status.cpp`

Poll Everywhere

pollev.com/tqm

```
int main(int argc, char* argv[]) {
  // fork a process to exec clang
  pid_t clang_pid = fork();

  if (clang_pid == 0) {
    // we are the child
    array<const char*, 5> argv = {
      "clang-15", "-o", "hello","hello_world.c", nullptr
    };
    execvp(argv.at(0), const_cast<char**>(argv.data()));
    exit(EXIT_FAILURE);
  }
  // fork to run the compiled program
  pid_t hello_pid = fork();
  if (hello_pid == 0) {
    // the process created by fork
    array<const char*, 2> argv {"./hello", nullptr};
    execvp(argv.at(0), const_cast<char**>(argv.data()));
    exit(EXIT_FAILURE);
  }
  wait(NULL);  // previously before second fork()
  wait(NULL);
  return EXIT_SUCCESS;
}
```

We take our previous code that we fixed and modify it. Now we call wait twice at the end of the program.

What happens?

Does our code still always work?

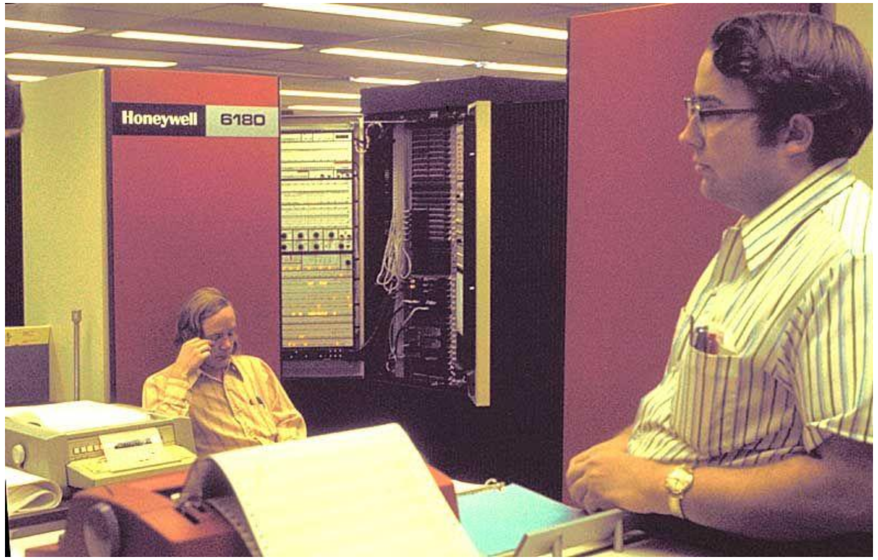
Lecture Outline

Multics: The Precursor

- **Multiplexed Information and Computing Service**

- Early time-sharing operating system
  - Time sharing: the sharing of a computer (mainframe) across multiple users at the same time
  - Necessary pre – personal computers (~1975)

- Started development in 1964
  - funded in part by Bell labs
- Bell Labs pulls out of Multics in 1969



"Unics"

- Ken Thompson and Dennis Ritchie lead the development of Unix
  - Both worked on Multics under Bell Labs

- Took some inspiration from Multics
  - Hierarchical file system
  - Text command line shell
  - The name:
    - Multics: Multiplexed Information and Computing Service
    - Unics: Uniplexed Information and Computing Service
    - At some point "Unics" became "Unix“
  - Unix rejected the overcomplexity of Multics

UNIX

- Originally (1970) was
  a singletasking system,
  without name or backing,
  and written in PDP assembly

- Functionality and multitasking added as other departments in Bell Labs needed them

- Departments kept adopting UNIX instead of built in OS's.
  - As a result, a support team was created, a UNIX Programmer's Manual was written, and man pages were created

UNIX and C

- B programming language by Ken Thompson
  - Was intended for writing UNIX utilities
- Dennis Ritchie modified B to make New B
  - Added things like types! (int, char, etc.)
- More features were added to New B, heavily influenced by its use in UNIX
- UNIX was soon re-written in C
  - One of the first operating systems (re)written in a higher-level-language (aka, not assembly)

Unix Adoption

- 1973: Unix was first presented formally outside of Bell Labs. Leading to many requests for the system

- Due to a 1956 decree, Bell System could not turn UNIX into a commercial product.
  - Bell had to license the product to anyone who asked
  - Code was “open source” of sorts.

- UNIX was continually updated, and C was as well.
  - Included the addition of pipes and other features
  - These updates made UNIX more portable to other systems.

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.

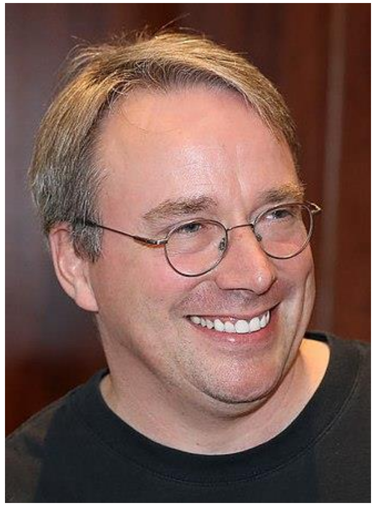
- Extra short version: "Keep it Simple, Stupid."

GNU

- In 1983, Bell Systems split up due to anti-trust laws.
  - A successor (AT&T) then turned UNIX into a commercial product, limiting rights to distribute/change/adapt/etc. UNIX

- Later that year, GNU is founded by Richard Stallman
  - GNU Not Unix
  - Copyleft
  - Goal: create a complete UNIX compatible system composed entirely of free software
  - Developed many required programs (libraries, editors, shell, compilers …) but missing low level elements like the kernel

Linux



- By 1991, a UNIX-like kernel that was Free Software did not exist
- Linus Torvalds was studying operating systems and wrote his own called Linux
  - This would be published under GPL 2 (GNU Public License)

- Blew up in popularity due to being free and open source

Unix-Like

- Almost all operating systems are UNIX related
  - “Genetically” related with historical connection to the original code base
  - Through the UNIX trademark once a system meets the Single UNIX Specification and is certified
  - Through “functionally” being UNIX-like. Behaving in a manner that is consistent with UNIX design and specification
    - Linux falls under this one

- Most Operating systems are Unix Like
  - Linux, macOS, iOS, Chrome OS, Android, etc.
  - Pretty much everything that is not Windows lol

Lecture Outline

Unix Shell

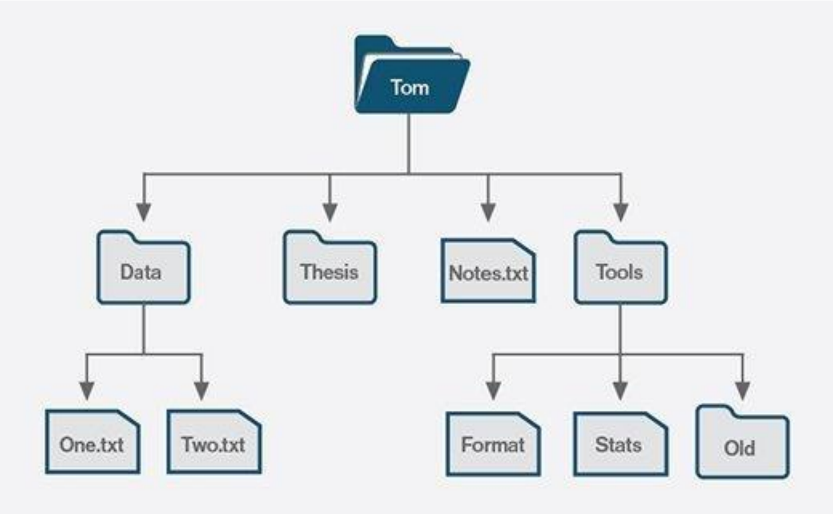
- A **user level** process that reads in commands
  - This is the terminal you use to compile, and run your code
- Commands can either specify one of our programs to run or specify one of the already installed programs
  - Other programs can be installed easily.
- There are many different shells, in 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.



. / ..

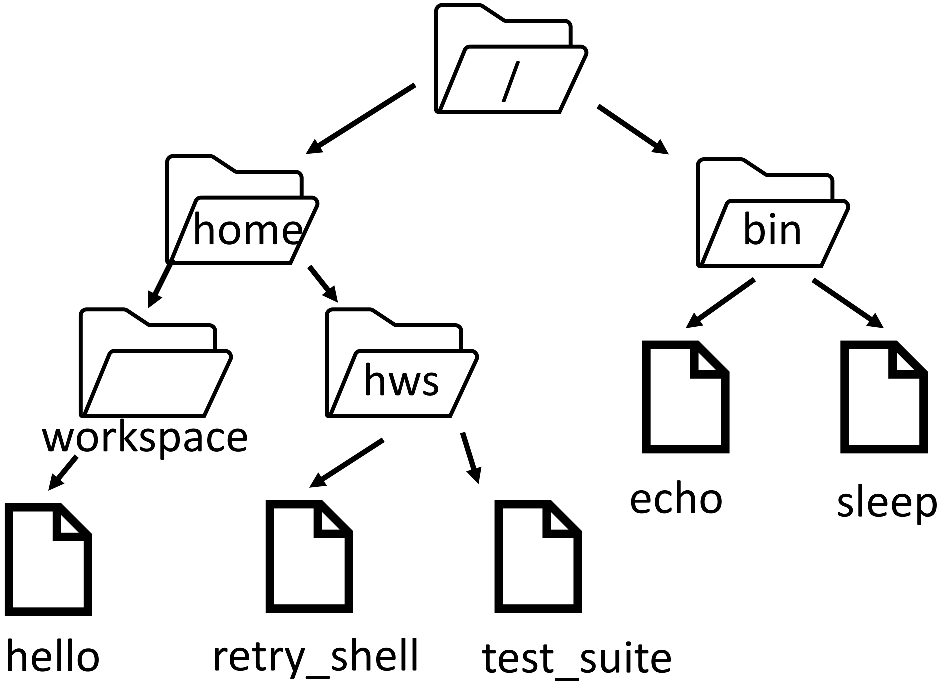
- "/" is used to connect directory and file names together to create a file path.
  - E.g. "**`workspace/595/hello/`**"
- "." is used to specify the current directory.
  - E.g. "**`./test_suite`**" tells to look in the current directory for a file called "**`test_suite`**"
- ".." is like "." but refers to the parent directory.
  - E.g. "**`./example/../test_suite`**" would be effectively the same as the previous example.

Poll Everywhere

pollev.com/tqm

- Are these valid paths to files? Assume that the current working directory is “home”

- ./test_suite
- /home/../bin/echo
- ../bin/sleep
- ./workspace/hello



Common Commands (Pt. 1)

- "**`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)

- "**`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)

- "**`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

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.

- These programs can be easily combined with UNIX Shell operators to solve more interesting problems **(More in a later lecture)**

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 check-time)
  - 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/

- YOU DO NOT NEED TO IMPLEMENT "echo" SPECIFICALLY
  - E.g. you should never have to check to see if user input contains the word "echo" in retry_shell. Just fork-exec the process.

retry_shell Demo

- In HW4, you will be writing your own shell that reads from user input
  - Each line is a command that could consist of a programs and it’s command line args
  - Your shell should fork a process to run each program and also support a “retry” feature”

- Some sample programs provided to help with implementation ideas.

- Also demo: /bin/

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++

Lecture Outline

Aside: File I/O & Disk

- File System:
  - Provides long term storage of data:
    - Persist after a program terminates
    - Persists after computer turns off
  - Data is organized into files & directories
    - A directory is pretty much a “folder”
  - Interaction with the file system is handled by the operating system and hardware. (To make sure a program doesn’t put the entire file system into an invalid state)

C Standard Library I/O

- In 5930, you've seen the C standard library to access files
  - Use a provided `FILE*` *stream* abstraction
  - **`fopen`**`()`, **`fread`**`()`, **`fwrite`**`()`, **`fclose`**`()`, **`fseek`**`()`

- These are convenient and portable
  - They are buffered*
  - They are implemented using lower-level OS calls

ALL FILE I/O IS BUILT ON TOP OF LOWER-LEVEL OS CALLS

From C to POSIX

- Most UNIX-en support a common set of lower-level file access APIs: POSIX – Portable Operating System Interface
  - **`open`**`()`, **`read`**`()`, **`write`**`()`, **`close`**`()`, **`lseek`**`()`
    - Similar in spirit to their `f*()` counterparts from the C std lib
    - Lower-level and unbuffered compared to their counterparts
    - Also less convenient
  - C and C++ stdlib doesn't provide everything POSIX does
    - You will have to use these to read file system directories and for network I/O, so we might as well learn them now

`open()`/`close()`

- To open a file:
  - Pass in the filename and access mode
  - Get back a "file descriptor"
    - Similar to `FILE*` from **`fopen()`**, but is just an `int`
      - Returns `-1` to indicate error
    - Must manually close file when done ☹

Used to identify
a file w/ the OS

```
#include <fcntl.h>     // for open()
#include <unistd.h>    // for close()
  ...
  int fd = open("foo.txt", O_RDONLY);
  if (fd == -1) {
    perror("open failed");
    exit(EXIT_FAILURE);
  }
  ...
  close(fd);
```

Reading from a File

Stores read result in buf

Number of bytes

- `ssize_t read(int fd, void* buf, size_t count);`

signed

  - Function is written in C: follows C design
    - Takes in a file descriptor
    - Takes in an array and length of where to store the results of the read
    - Returns number of bytes read
  - EVERY TIME we read from a file, this function is getting called somewhere
    - Even in Java or Python
    - There are wrappers around this, but they are all implemented on top of these system calls
    - The OS doesn't speak java or python, it "speaks" assembly and C so all languages must have a way to invoke these C functions.

Going over this quickly: the important point is not to memorize this function; we will go over it again later.

The main thing is this: whenever we interact with a file (even in other languages) somewhere under the hood it is calling these C functions

Example Read Code

```
int fd = open(filename, O_RDONLY);
array<char, 1024> buf {};  // buffer of appropriate size
ssize_t result;

result = read(fd, buf.data(), 1024);
if (result == -1) {
  // an error happened, so exit the program
  // print out some error message to cerr
  exit(EXIT_FAILURE);
}

// If we want to construct a string from the bytes read
// we need to say how many bytes to take from the array.
string data_read(buf.data(), result);

// Whenever we are done with the file, we must close it
close(fd);
```

Going over this quickly: the important point is not to memorize this function; we will go over it again later.

The main thing is this: whenever we interact with a file (even in other languages) somewhere under the hood it is calling these C functions

Lecture Outline

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

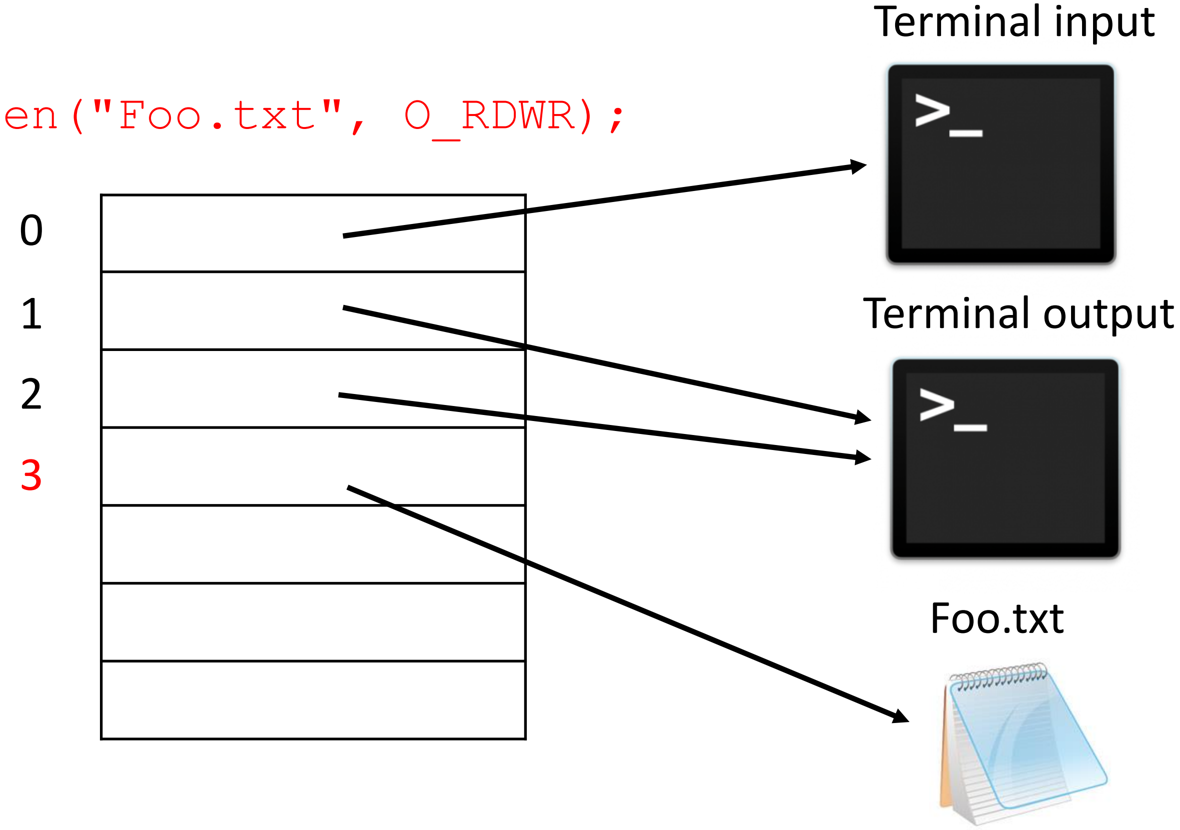
stdout, stdin, stderr

- 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

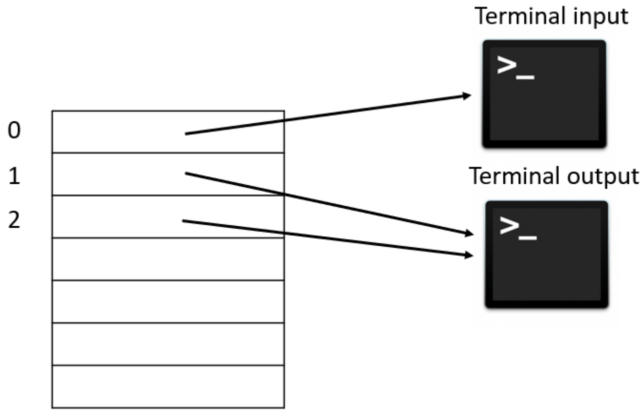
- In addition to an address space, each process will have **its own file descriptor table** managed by the OS
- The table is just an array, and the file descriptor is an index into it.

`open("Foo.txt", O_RDWR);`



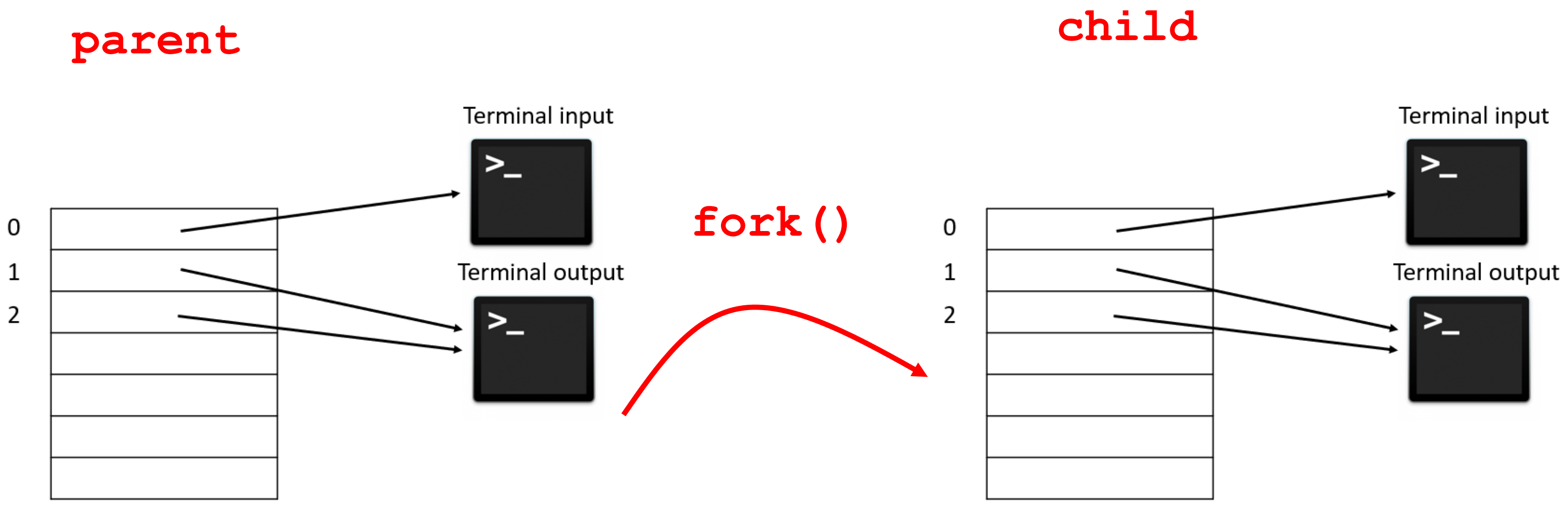
File Descriptor Table: Per Process

- 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



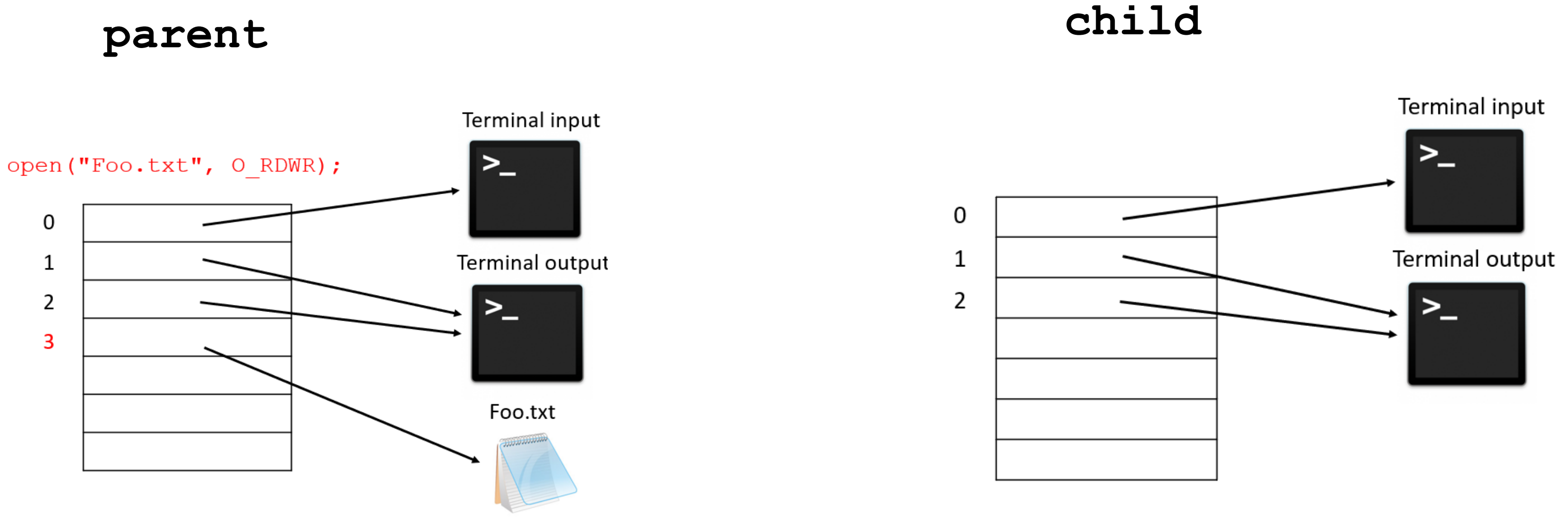
File Descriptor Table: Per Process

- 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!

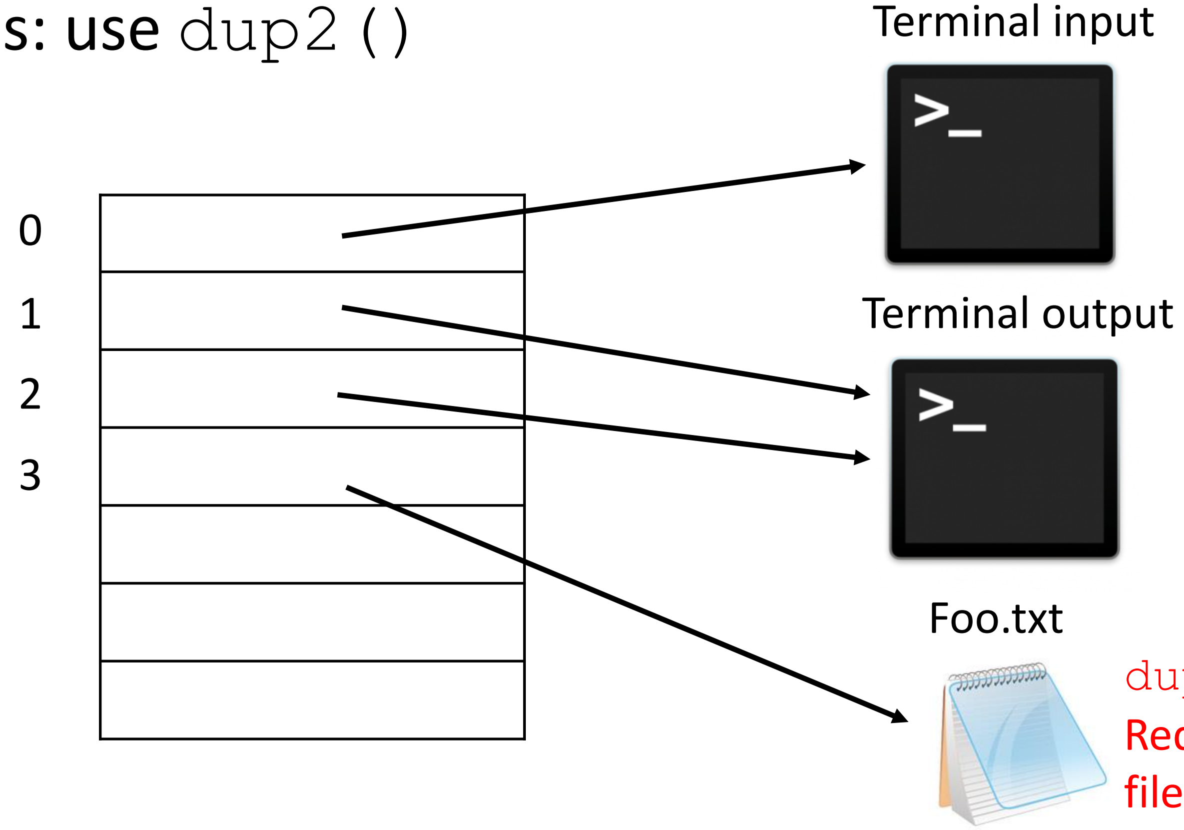
Gap Slide

- 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

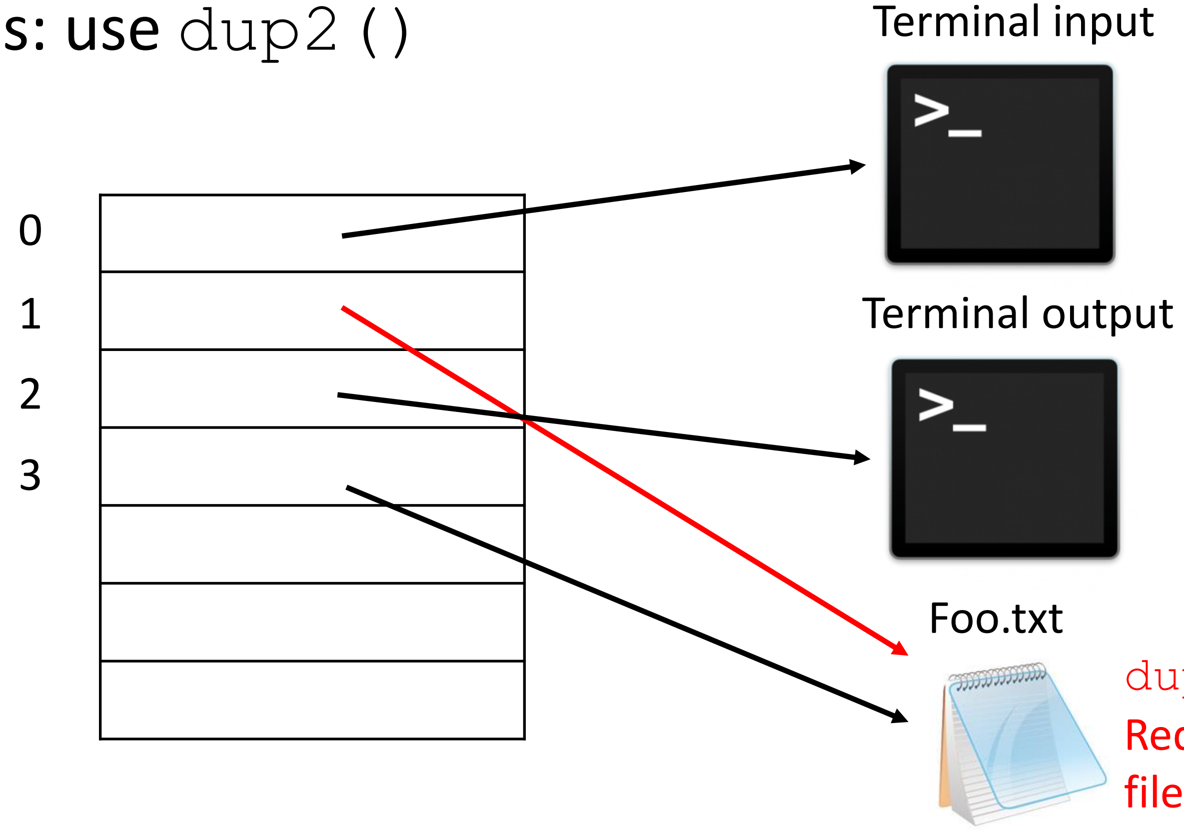
- We can change things so that STDOUT_FILENO is associated with something other than a terminal output.
- Now, any calls to printf, cout, System.out, etc now go to the redirected output
- To do this: use `dup2()`



`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()`

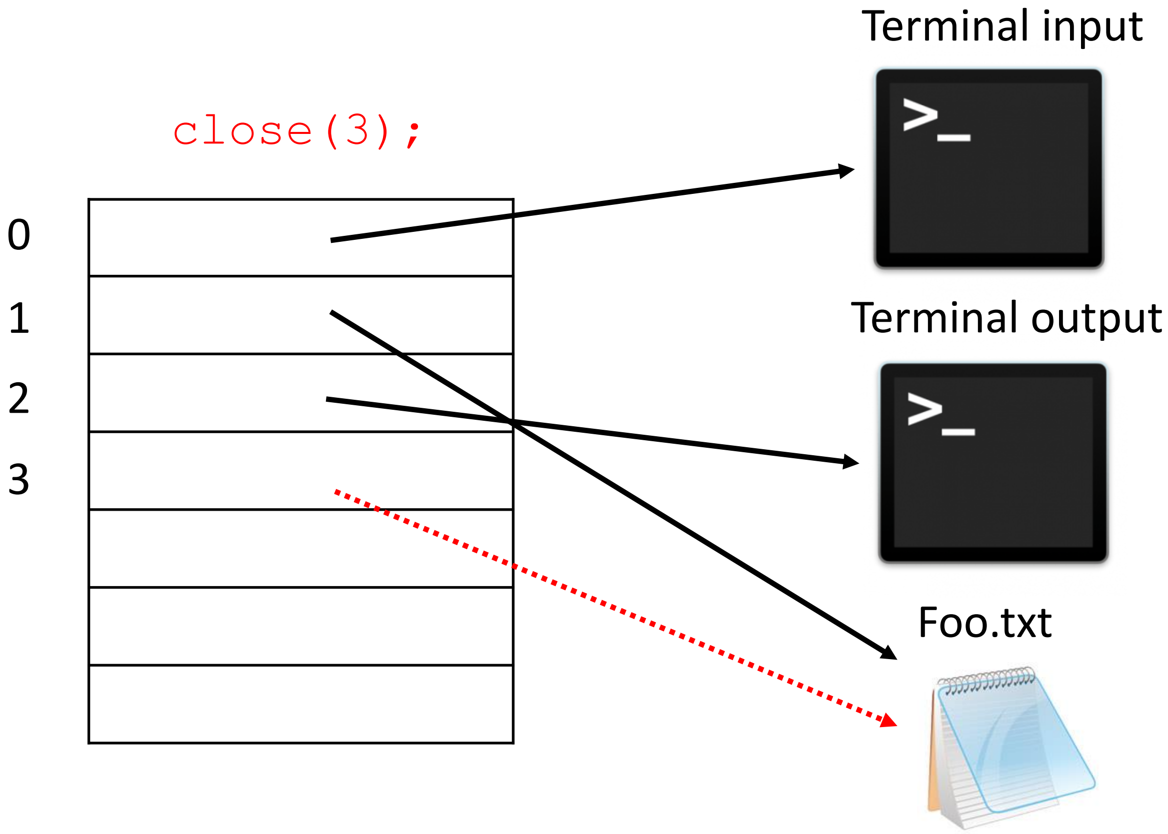


`dup2(3, STDOUT_FILENO);`

Redirects stdout to go to file descriptor 3's destination

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);`

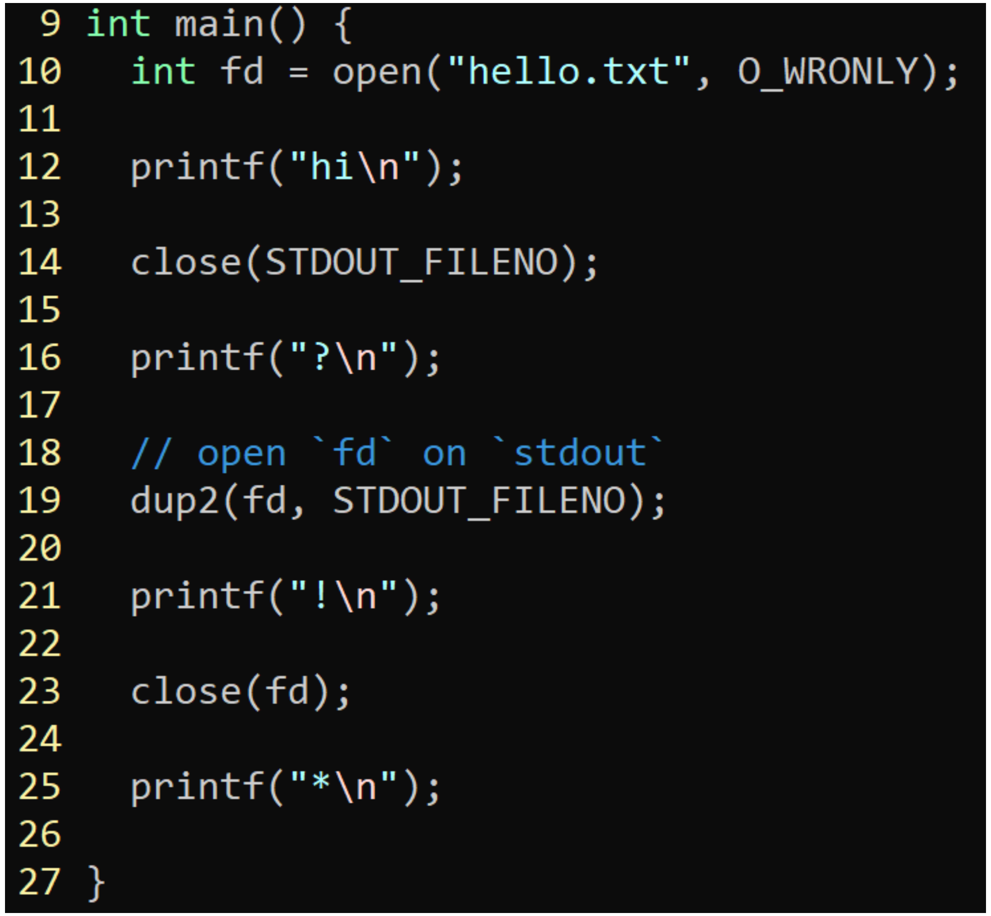
File descriptor

  - 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.

Poll Everywhere

pollev.com/tqm

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



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

  printf("hi\n");

  close(STDOUT_FILENO);

  printf("?\n");

  // open `fd` on `stdout`
  dup2(fd, STDOUT_FILENO);

  printf("!\n");

  close(fd);

  printf("*\n");

}
```

Lecture Outline

Pipes

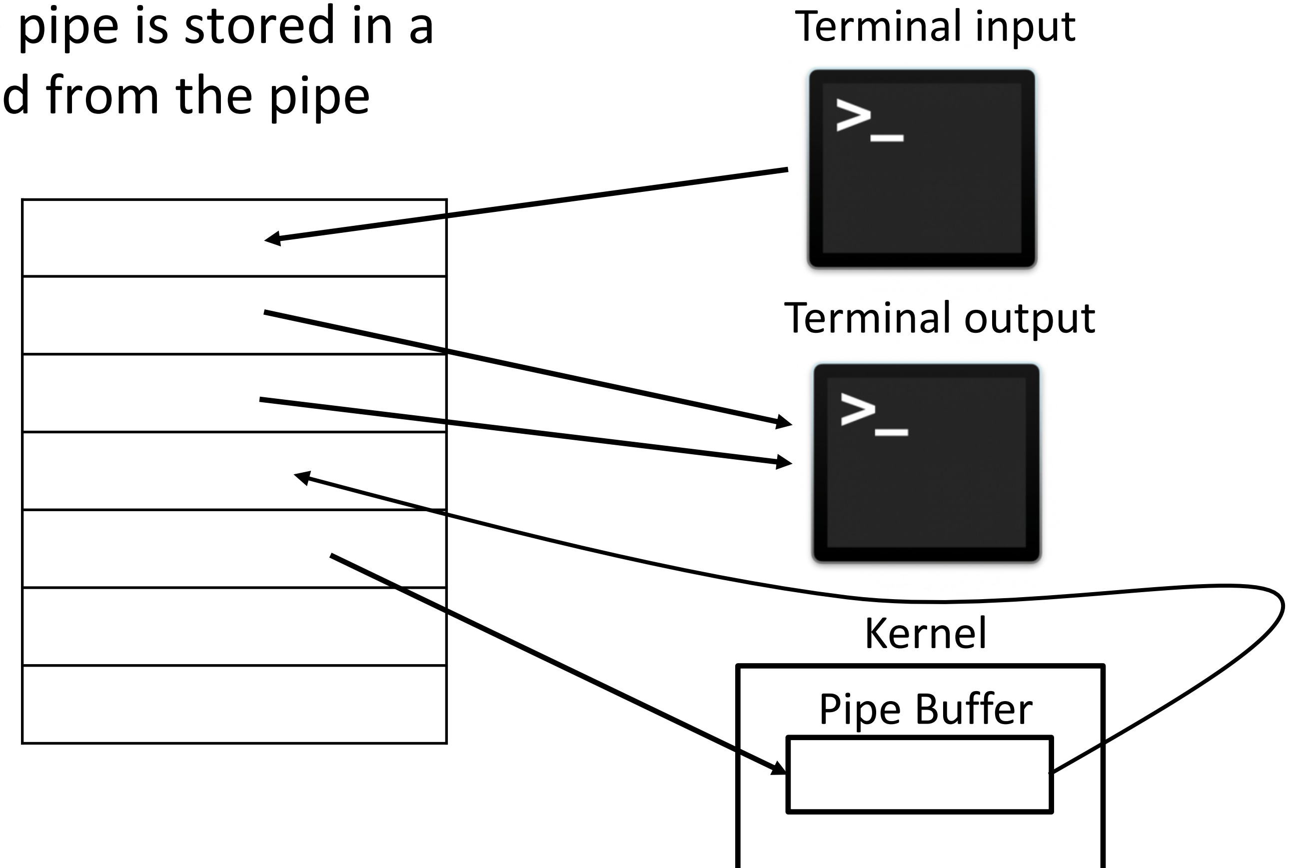
```
int pipe(int pipefd[2]);
```

- 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

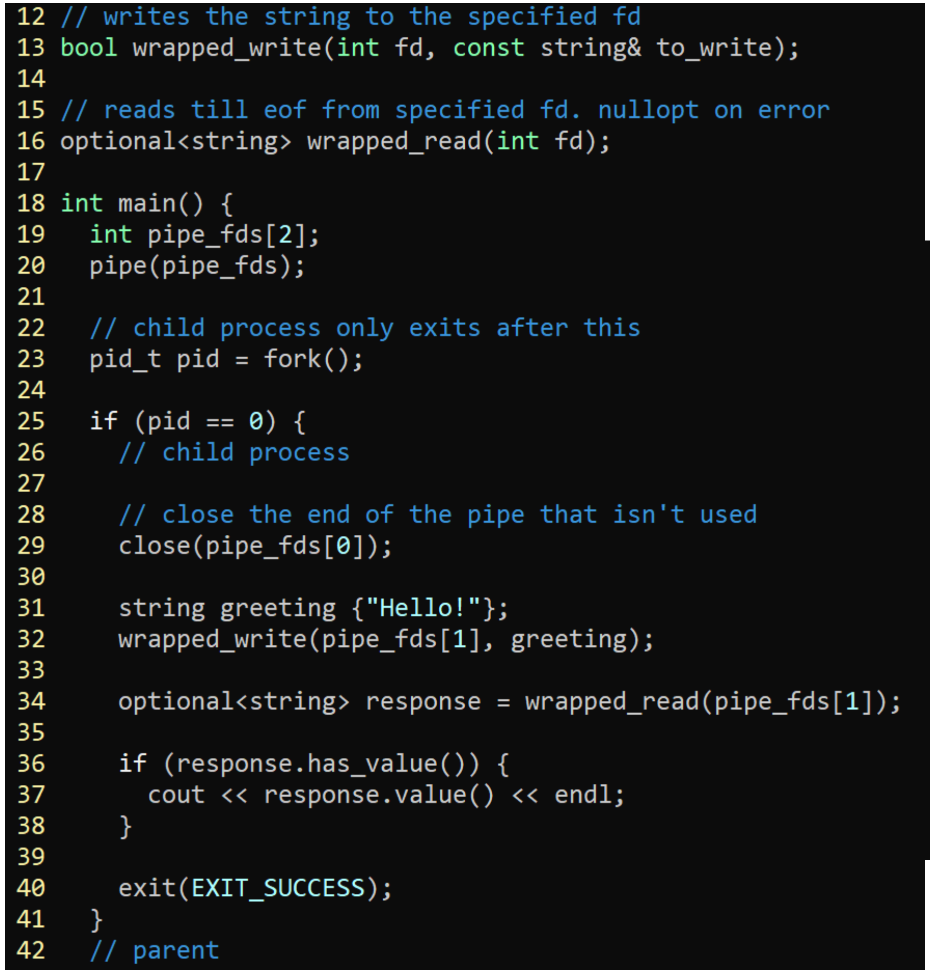
- EOF is only read from a pipe when:
  - There is nothing in the pipe
  - All write ends of the pipe are closed

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

Poll Everywhere

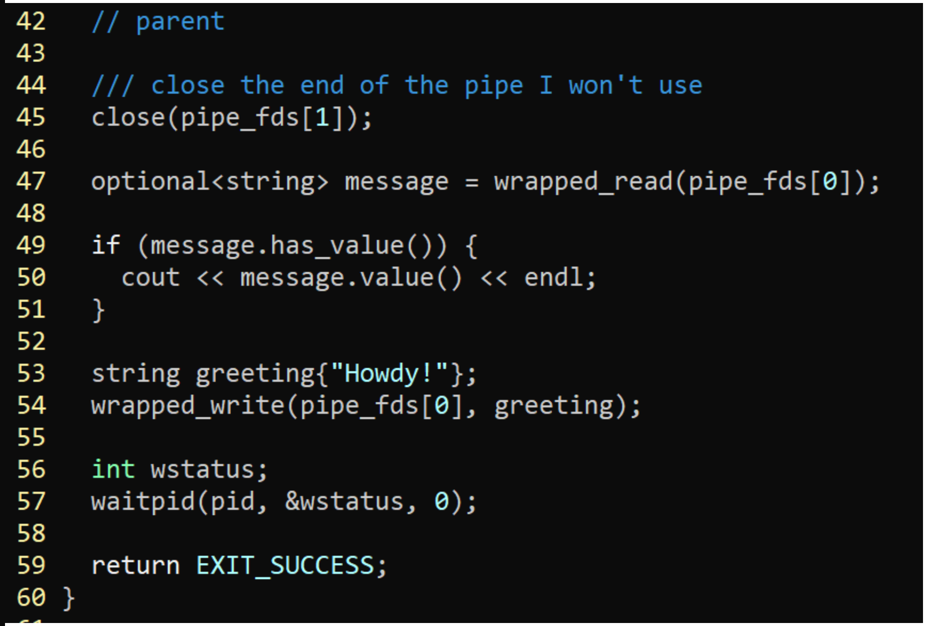
pollev.com/tqm

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



```
// writes the string to the specified fd
bool wrapped_write(int fd, const string& to_write);

// reads till eof from specified fd. nullopt on error
optional<string> wrapped_read(int fd);

int main() {
  int pipe_fds[2];
  pipe(pipe_fds);

  // child process only exits after this
  pid_t pid = fork();

  if (pid == 0) {
    // child process

    // close the end of the pipe that isn't used
    close(pipe_fds[0]);

    string greeting {"Hello!"};
    wrapped_write(pipe_fds[1], greeting);

    optional<string> response = wrapped_read(pipe_fds[1]);

    if (response.has_value()) {
      cout << response.value() << endl;
    }

    exit(EXIT_SUCCESS);
  }
  // parent
```

pipe_unidirect.cpp
on course website



```
  // parent

  /// close the end of the pipe I won't use
  close(pipe_fds[1]);

  optional<string> message = wrapped_read(pipe_fds[0]);

  if (message.has_value()) {
    cout << message.value() << endl;
  }

  string greeting{"Howdy!"};
  wrapped_write(pipe_fds[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

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

That’s it for now!

- More next lecture ☺
- Especially more on pipes()