C interop & Processes

Computer Systems Programming, Spring 2024

Instructor: Travis McGaha

TAs:

Ash Fujiyama
CV Kunjeti
Felix Sun
Heyi Liu
Kevin Bernat
Lang Qin
Sean Chuang
Serena Chen
Yuna Shao

Logistics

- Exam grades posted Monday night
  - Regrade requests opened 24 hours after grades are posted
  - Will be open for a week (Tuesday 4/9 @ 11:59pm)
  - Rember that we have the clobber policy, it is ok if the exam did not go well.
- HW03 released: due Friday next week
  - Recitation tomorrow will be helpful for understanding it conceptually
- Project to be posted soon
  - Partner sign up released Monday, due on Friday
  - Code and specification posted soon
- Checkin to be released soon

pollev.com/tqm

- Any questions?

Lecture Outline

- **C++ & C Interop**
  - **C Strings & Arrays w/ C++**
- Processes & Fork
- stdin, stdout, stderr & File Descriptors
- Exec
- Pipe

POSIX

- POSIX – Portable Operating System Interface
  - Supported on most operating systems
  - Provides access to many features that are not available directly from the C or C++ standard library
    - If a language does support something that POSIX provides, it almost certainly is done by calling these system calls
    - Example: `open()`, `read()`, `write()`, `close()`, `lseek()`
- POSIX is implemented in C
  - This means if any language wants to take advantage of these features, it must know how to call C code
  - C++ can interact with C code directly, with only a few changes

C Arrays

- Definition: `type name[size]`
  - Allocates `size*sizeof(type)` bytes of *contiguous* memory
  - Normal usage is a compile-time constant for `size` (*e.g.* `int scores[175];`)
  - Initially, array values are "garbage"

- Size of an array
  - **Not stored anywhere** – array does not know its own size!
  - The programmer will have to store the length in another variable or hard-code it in
  - ***Sometimes*** can store `nullptr` or a special value to mark the end of an array.

Pointer Arithmetic

- We can treat pointers as if they are C-style arrays.
  - Note: not every pointer is necessarily an Array.
    - An `int*` could point to an array of integers or only one integer
    - In either case the "**`arr[i]`**" syntax will compile for all pointers.

```
int a[] = {0, 3, 5, 9};
int size = 4;

int sum = 0;
int* ptr = a; // &(a[0])
for (int i = 0; i < size; i++) {
  sum += ptr[i];
}
```

C++ Arrays

- C arrays are considered dangerous, and not safe to use
  - Length is not attached to the array
  - There is no bounds checking
  - Arrays are not readable code
    Consider this CIS 5480 Example:
    What do you think "commands" represents?

```
// example from CIS 5480
struct parsed_command {
  int num_commands;
  char*** commands;
};
```

- In our code, we will use C++ Arrays instead, but we need to call C code that expects C arrays…

C++ Arrays -> C array

- Can use **.data**() and **.size**() to convert to a C array

```
int sumAll(int* a, int size) {
  int i, sum = 0;
  for (i = 0; i < size; i++) {
    sum += a[i];
  }
  return sum;
}

int main(){
  array<int, 1024> arr{};
  sumAll(arr.data(), arr.size());
}
```

C++ Vector

- C++ Vector is a dynamically re-sizeable array
  - If we need more (or less) elements, the "array" can grow or shrink to accommodate for this
  - C++ vector is implemented as an object that is just a wrapper around a C-style array that is allocated on the heap.
    - The internal C-style array is re-allocated whenever we need more space.

C Strings (without Objects)

- Strings are central to C, very important for I/O
- In C, we don't have Objects but we need strings
- If a string is just a sequence of characters, we can have use array of characters as a string

- Example:

```
char str_arr[] = "Hello World!";
char *str_ptr = "Hello World!";
```

C-string "end" (Null Termination)

- Arrays don't have a length, but we **mark the end of a string with the null terminator character.**
  - The null terminator has value **`0x00`** or **`'\0'`**
  - Well formed strings ***MUST*** be null terminated

- Example: `char str[] = "Hello";`
  - Takes up 6 characters, 5 for "Hello" and 1 for the null terminator

address	0x2000	0x2001	0x2002	0x2003	0x2004	0x2005
value	'H'	'e'	'l'	'l'	'o'	'\0'

- **`strlen()`** takes in a c-string and returns the length (not counting the null-terminator)

C++ Strings

- C++ std::string is just an object that manages ("wraps around") a C-string. Reallocating when necessary.

```
class string {
 public:
  string(const char* c_string) {
    length_ = strlen(c_string);
    capacity_ = length_ + 1;
    data_ = new char[capacity_];
    for (size_t i = 0; i <= length_; i++) {
      data_[i] = c_string[i]
    }
  }

 private:
  char* data_;
  size_t capacity_;
  size_t length_;
};
```

C++ Strings -> C Strings

- C++ Strings can grant access to the underlying C-String through the function `.c_str()`

- This is useful for when interfacing with C code from C++:

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

Poll Everywhere

pollev.com/tqm

- What does this code print?

```
int mystery(int* a, int size) {
  int sum = 0;
  for (size_t i = 0; i < size; i++) {
    sum += a[i];
    a[i] = sum;
  }
  return sum;
}

int main(){
  vector<int> vec{3, 4};
  mystery(vec.data(), vec.size());
  vec.push_back(5);
  for (auto& n : vec) {
    cout << n << endl;
  }
}
```

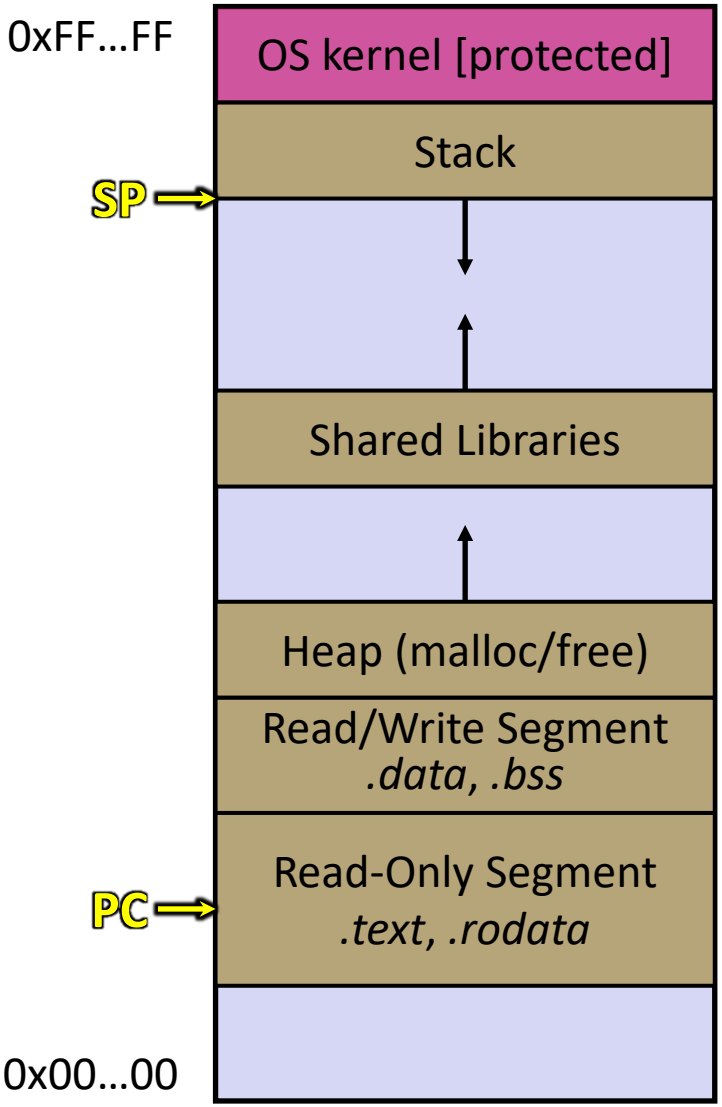
Lecture Outline

- C++ & C Interop
- **Processes & Fork**
- stdin, stdout, stderr & File Descriptors
- Exec
- Pipe

Review: Address Spaces

- A process has its own *address space*
  - Includes segments for different parts of memory
  - A process usually has one or more threads
    - A thread tracks its current state using the stack pointer (SP) and program counter (PC)
- New processes are created with:

```
pid_t fork();
```

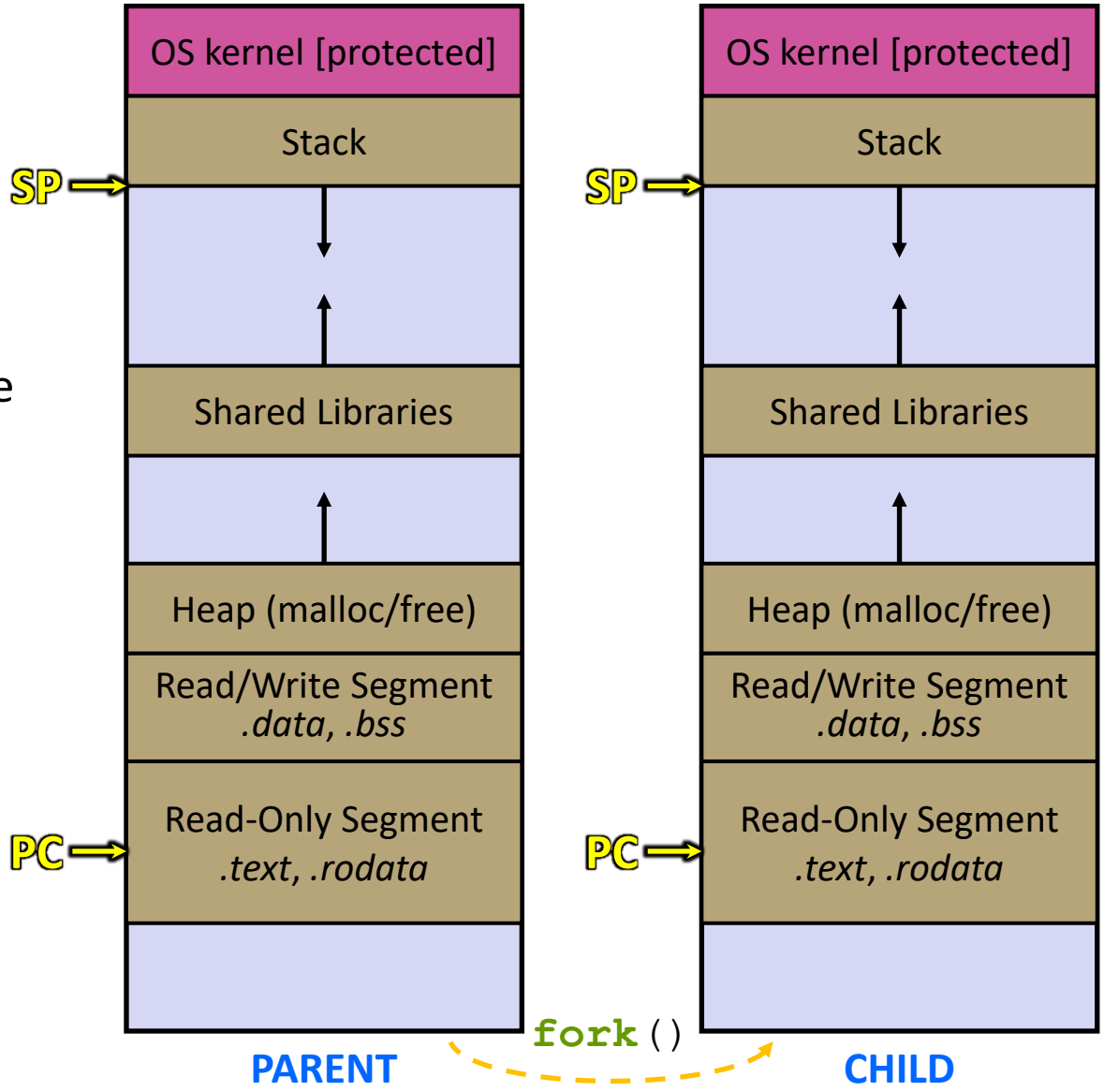


Creating New Processes

- `pid_t fork();`
  - Creates a new process (the "child") that is an *exact clone** of the current process (the "parent")
    - *almost everything
  - The new process has a separate virtual address space from the parent

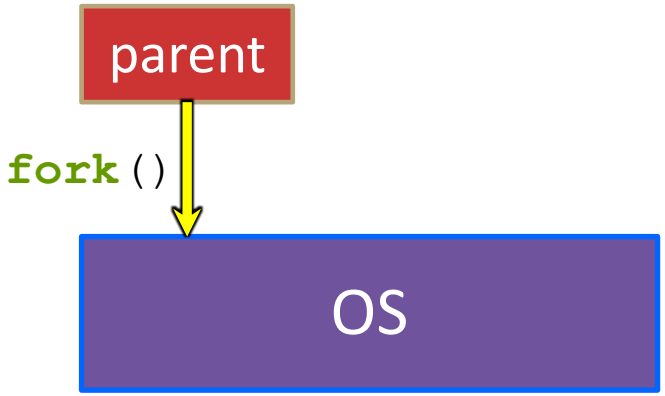
`fork()` and Address Spaces

- Fork causes the OS to clone the address space
  - The *copies* of the memory segments are (nearly) identical
  - The new process has *copies* of the parent's data, stack-allocated variables, open file descriptors, etc.



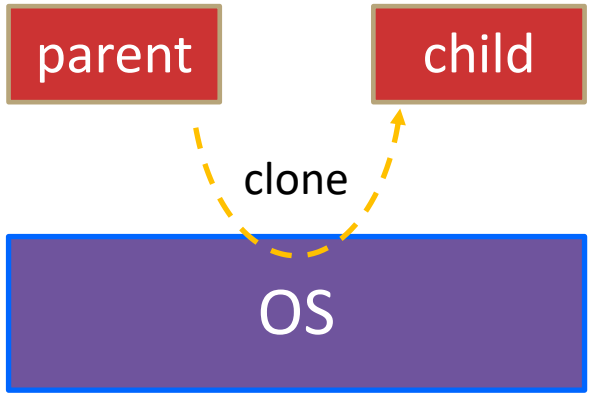
`fork()`

- `fork()` has peculiar semantics
  - The parent invokes `fork()`
  - The OS clones the parent
  - *Both* the parent and the child return from fork
    - Parent receives child's pid
    - Child receives a 0



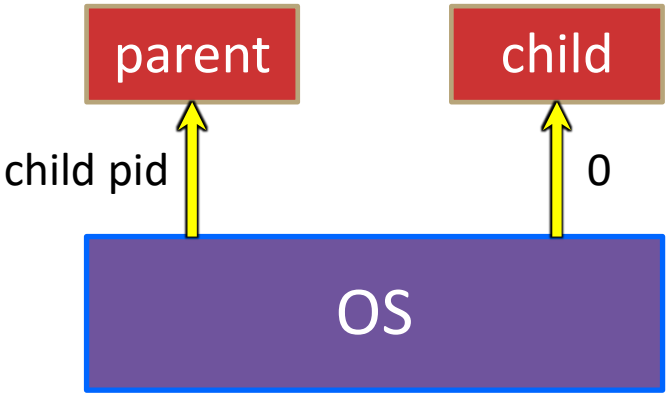
`fork()`

- `fork()` has peculiar semantics
  - The parent invokes `fork()`
  - The OS clones the parent
  - *Both* the parent and the child return from fork
    - Parent receives child's pid
    - Child receives a 0



`fork()`

- `fork()` has peculiar semantics
  - The parent invokes `fork()`
  - The OS clones the parent
  - *Both* the parent and the child return from fork
    - Parent receives child's pid
    - Child receives a 0



fork() example

```
cout << "Hello!" << endl;
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 595;
} else {
  x = 593;
}
cout << x << endl;
```

Always prints "Hello"

fork() example

```
cout << "Hello!" << endl;
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 595;
} else {
  x = 593;
}
cout << x << endl;
```

Always prints "Hello"

fork() example

Parent Process (PID = X)

```
cout << "Hello!" << endl;
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 595;
} else {
  x = 593;
}
cout << x << endl;
```

fork_ret = Y

Child Process (PID = Y)

```
cout << "Hello!" << endl;
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 595;
} else {
  x = 593;
}
cout << x << endl;
```

fork_ret = 0

fork()

Always prints "Hello"

Does NOT print "Hello"

fork() example

Parent Process (PID = X)

```
cout << "Hello!" << endl;
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 595;
} else {
  x = 593;
}
cout << x << endl;
```

Child Process (PID = Y)

```
cout << "Hello!" << endl;
pid_t fork_ret = fork();
int x;

if (fork_ret == 0) {
  x = 595;
} else {
  x = 593;
}
cout << x << endl;
```

fork_ret = Y

fork()

fork_ret = 0

Always prints "Hello"

Always prints "593"

Always prints "595"

Exiting a Process

- `void exit(int status);`
  - Causes the current process to exit normally
  - Automatically called by **`main()`** when main returns
  - Exits with a return status (e.g. EXIT_SUCCESS or EXIT_FAILURE)
    - This is the same int returned by **`main()`**
  - The exit status is accessible by the parent process with **`wait()`** or **`waitpid()`**.

Poll Everywhere

pollev.com/tqm

```
int global_num = 1;

void function() {
  global_num++;
  cout << global_num << endl;
}

int main() {
  pid_t id = fork();

  if (id == 0) {
    function();
    id = fork();
    if (id == 0) {
      function();
    }
    return EXIT_SUCCESS;
  }

  global_num += 2;
  cout << global_num << endl;
  return EXIT_SUCCESS;
}
```

- How many numbers are printed? What number(s) get printed from each process?

pollev.com/tqm

- How many times is ":)" printed?

```
int main(int argc, char* argv[]) {
  for (int i = 0; i < 4; i++) {
    fork();
  }

  cout << ":)\n"; // "\n" is similar to endl
  return EXIT_SUCCESS;
}
```

"join"-ing a Process

- ```
  pid_t waitpid(pid_t pid, int *wstatus,
                int options);
  ```
  - The "process equivalent" of **`pthread_join()`**
  - 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
- ```
  pid_t wait(int *wstatus);
  ```
  - Equivalent of waitpid, but waits for ANY child

Demo: `fork_example`

- See `fork_example.cpp`
  - Brief code demo to see the various states of a process
    - Running
    - Zombie
    - Terminated
  - Makes use of `sleep()`, `waitpid()` and `exit()`!

Lecture Outline

- C++ & C Interop
- Processes & Fork
- **stdin, stdout, stderr & File Descriptors**
- Exec
- Pipe

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`

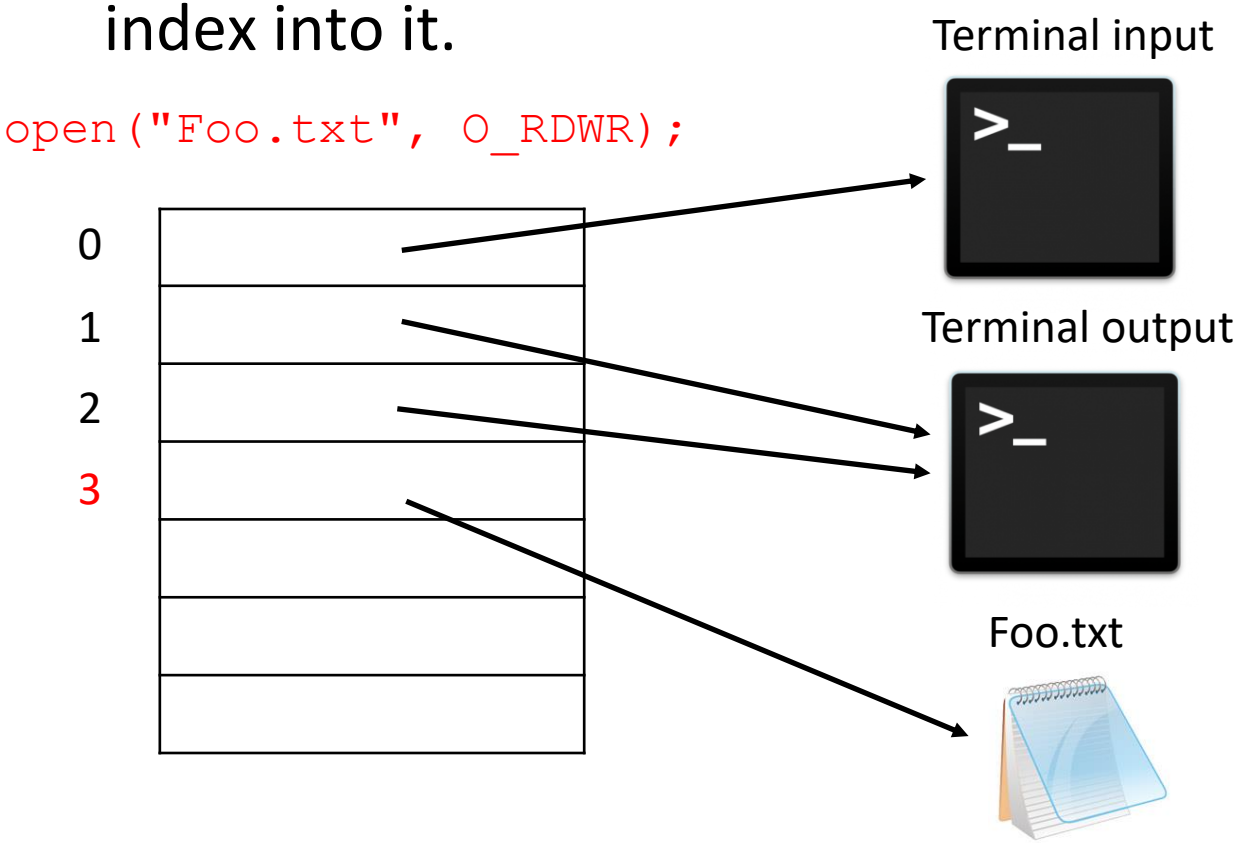
- These will be open on default for a process

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

File Descriptor Table

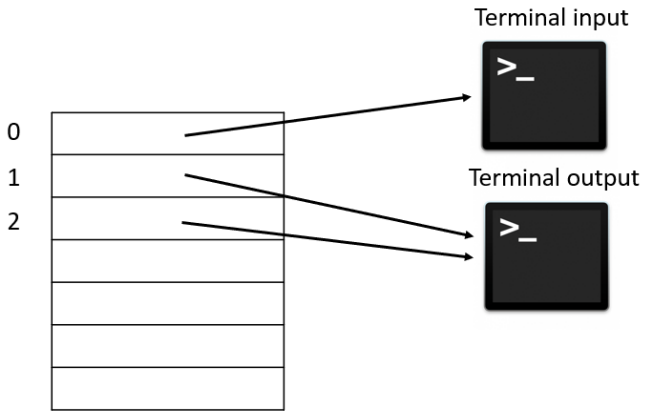
- In addition to an address space, each process will have **its own file descriptor table** managed by the OS
- The table is just an array, and the file descriptor is an index into it.

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



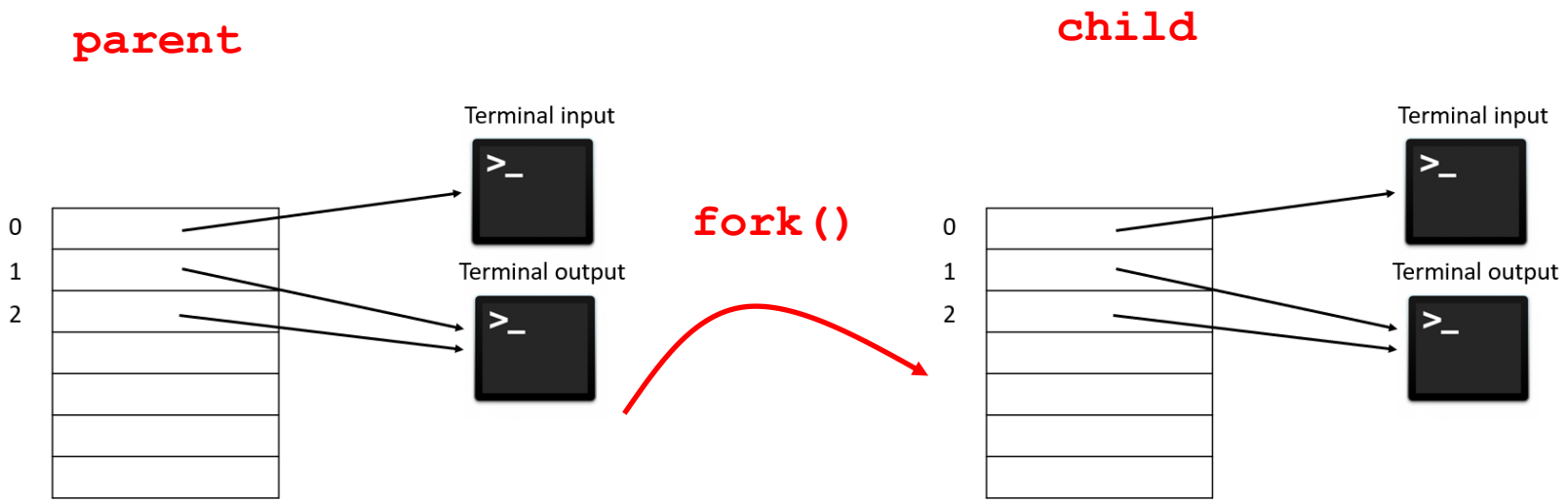
File Descriptor Table: Per Process

- 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

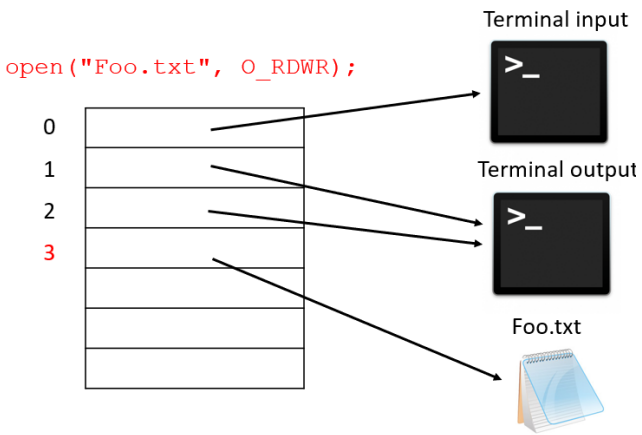
- each process will have **its own file descriptor table** managed by the OS
- Fork will make a copy of the parent's file descriptor table for the child



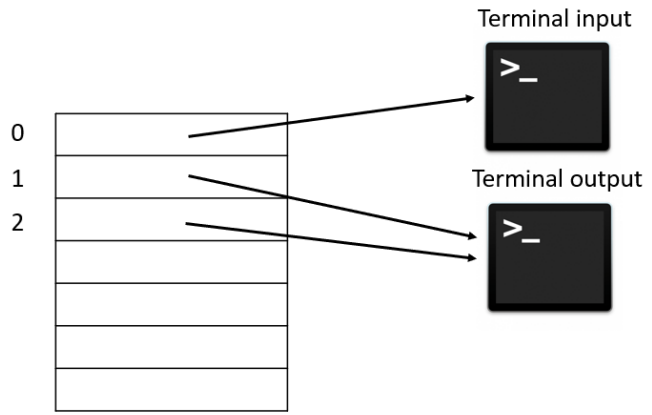
File Descriptor Table: Per Process

- 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

parent



child



Child is unaffected by parent calling open!

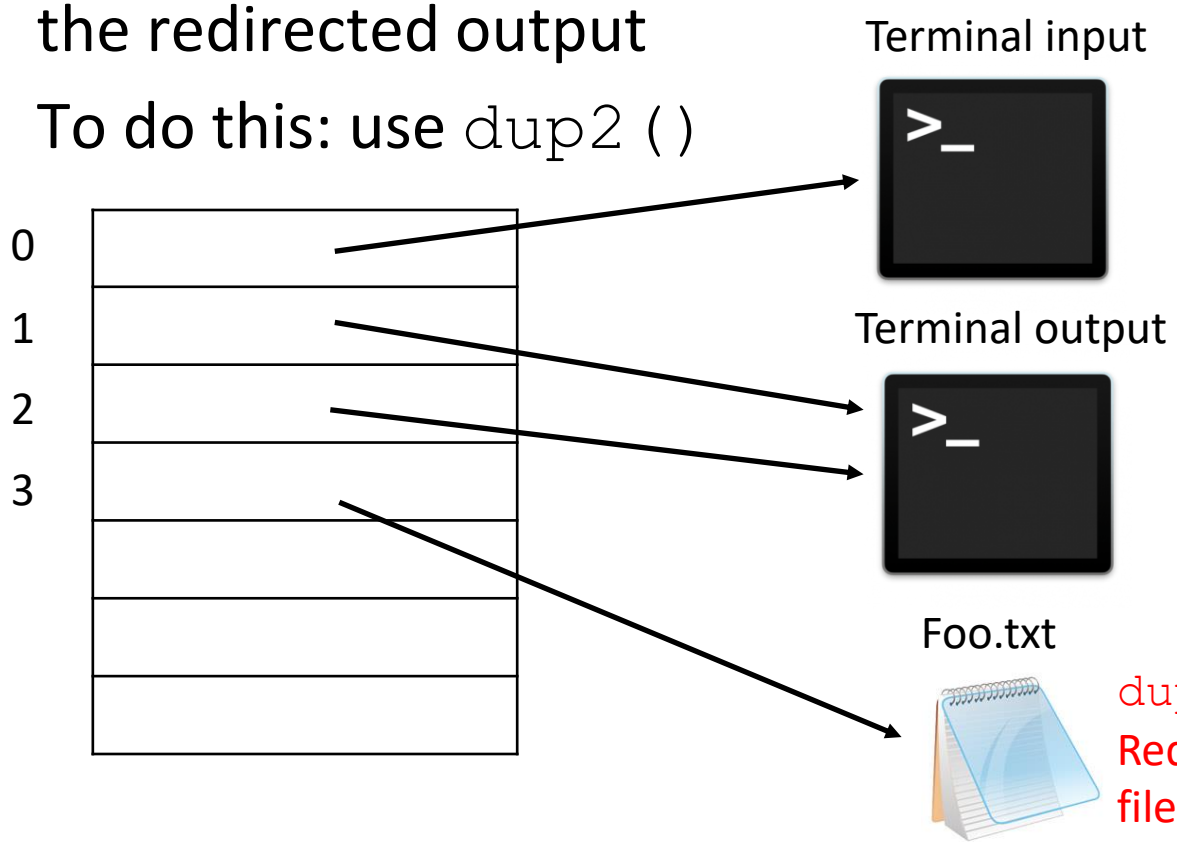
Gap Slide

- 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

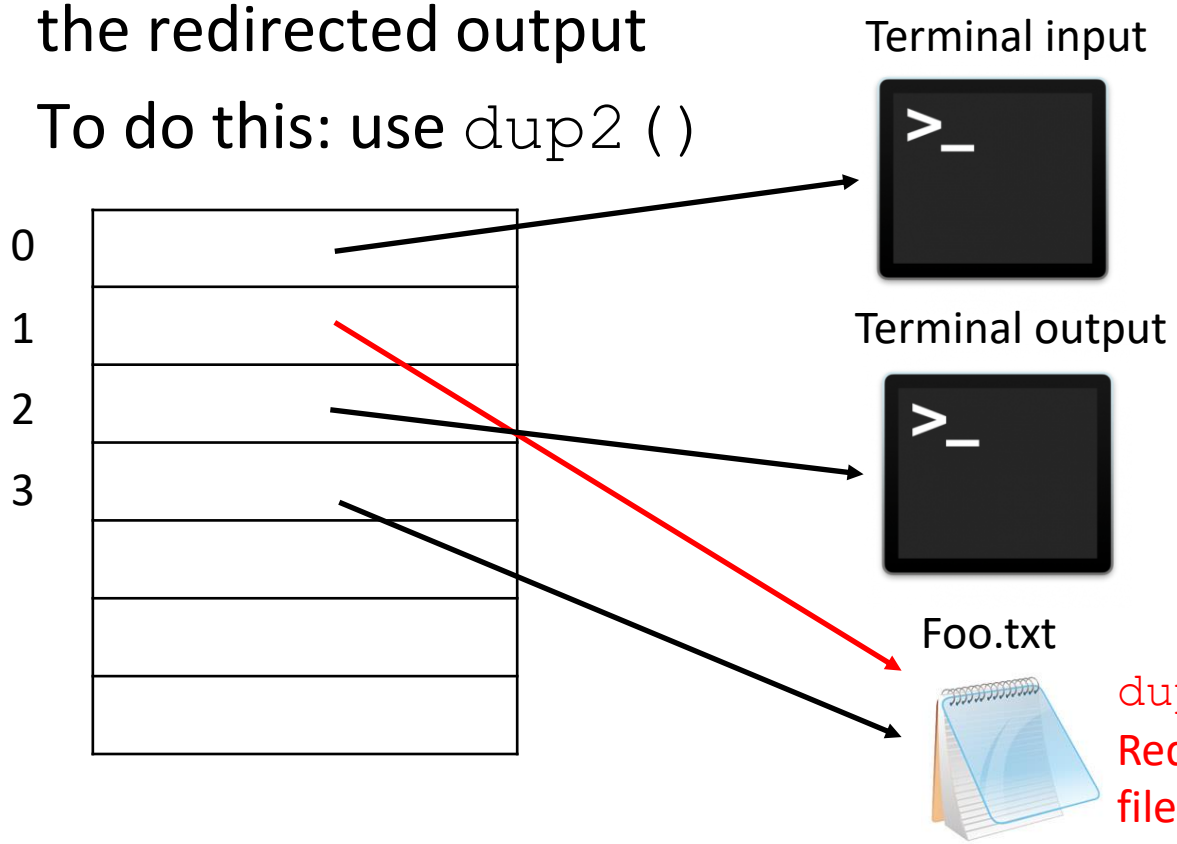
- We can change things so that STDOUT_FILENO is associated with something other than a terminal output.
- Now, any calls to printf, cout, System.out, etc now go to the redirected output
- To do this: use `dup2()`



`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()`

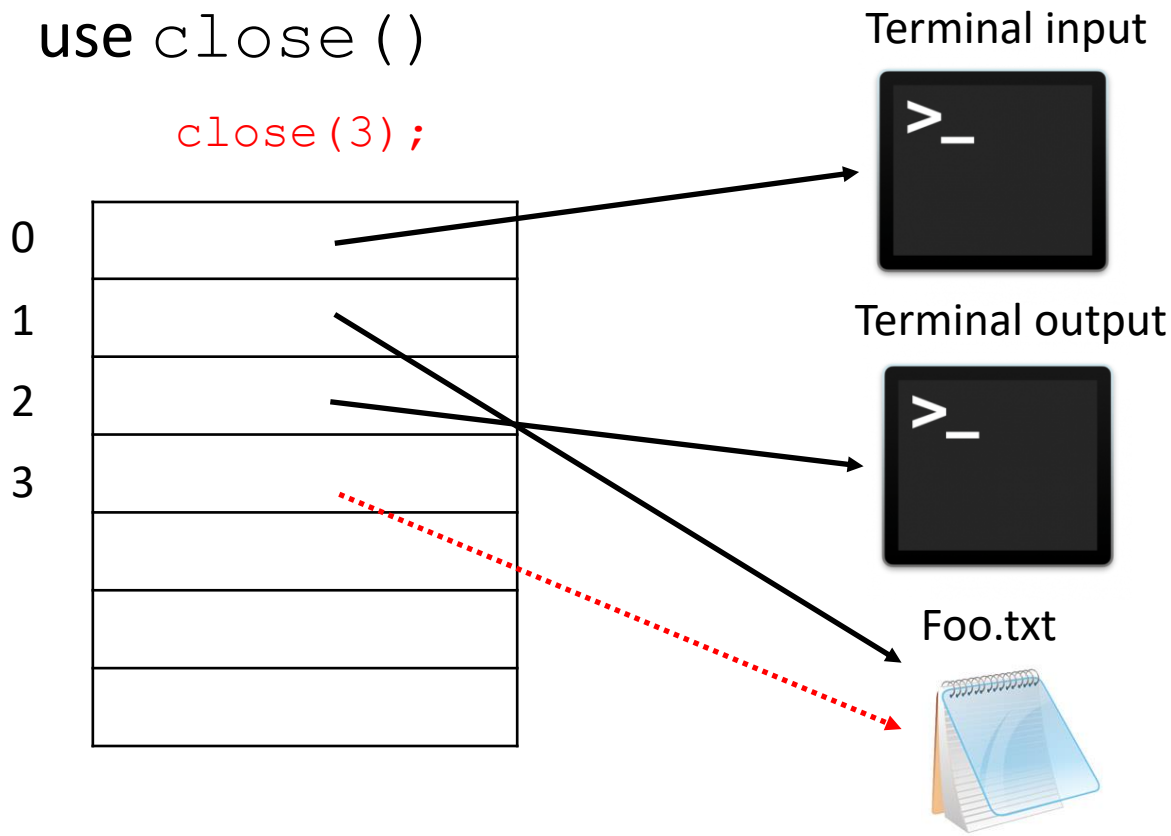



```
dup2(3, STDOUT_FILENO);
```

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()`



Poll Everywhere

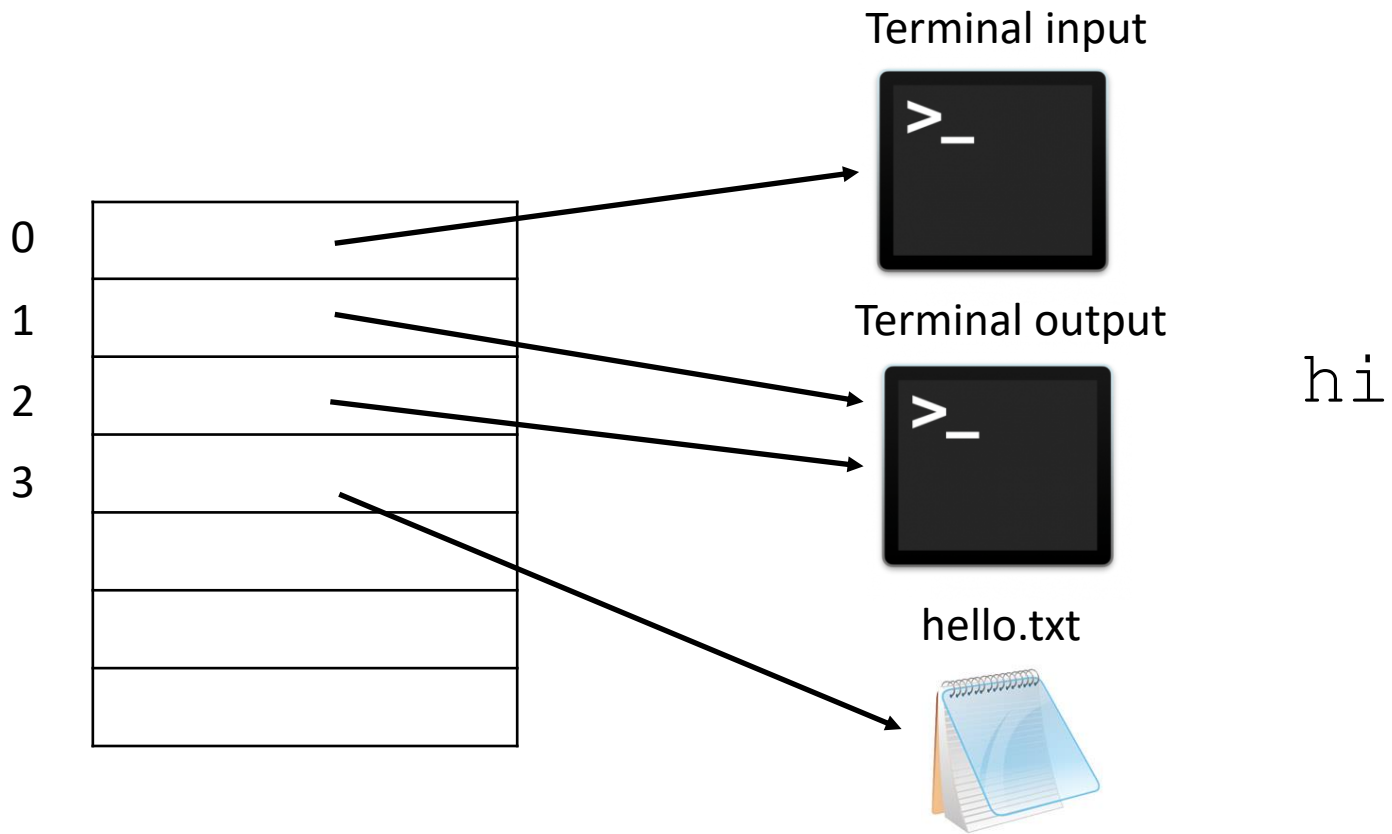
pollev.com/tqm

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

```
 9 int main() {
10   int fd = open("hello.txt", O_WRONLY);
11
12   printf("hi\n");
13
14   close(STDOUT_FILENO);
15
16   printf("?\n");
17
18   // open `fd` on `stdout`
19   dup2(fd, STDOUT_FILENO);
20
21   printf("!\n");
22
23   close(fd);
24
25   printf("*\n");
26
27 }
```

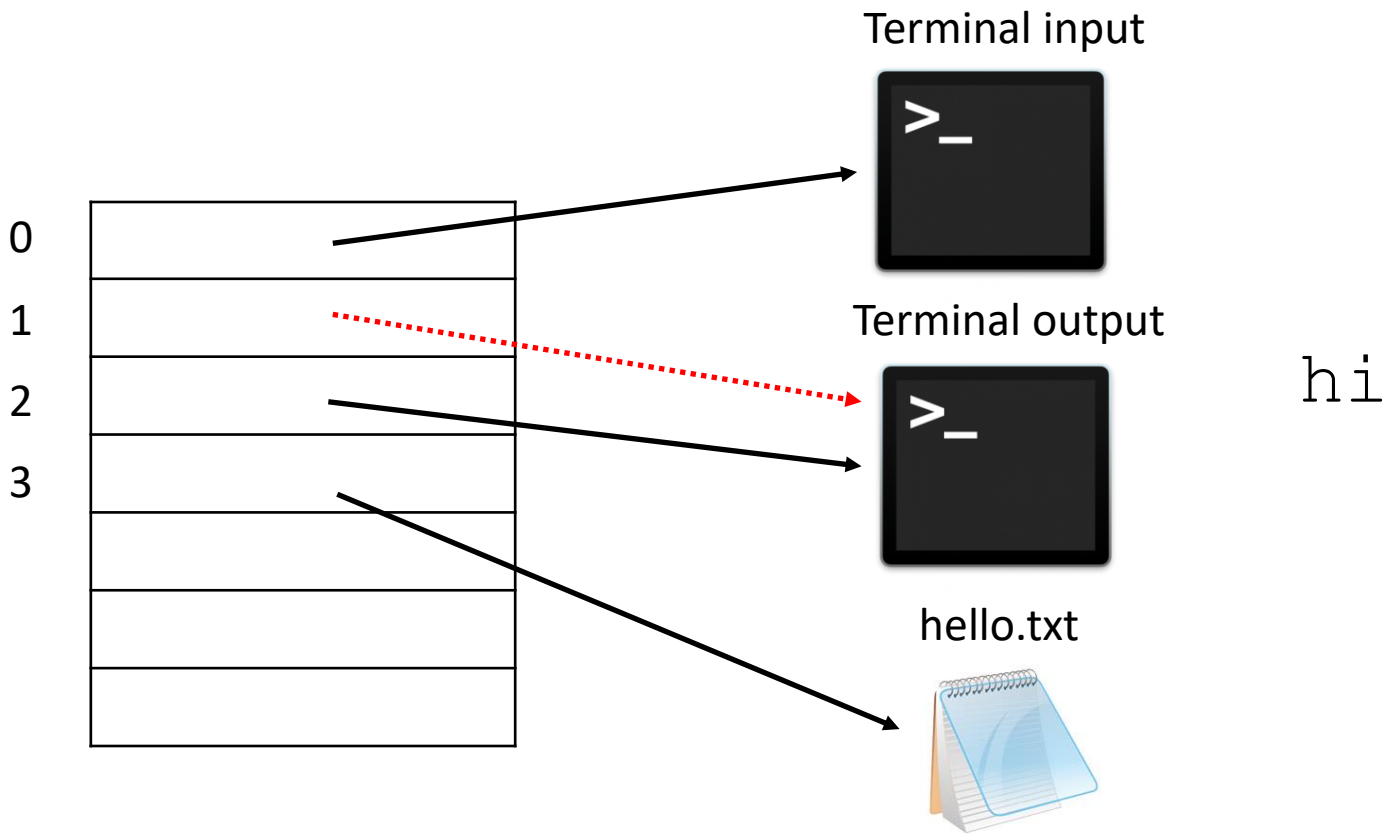
Explanation

```
int fd = open("hello.txt", O_WRONLY);
printf("hi\n");
```



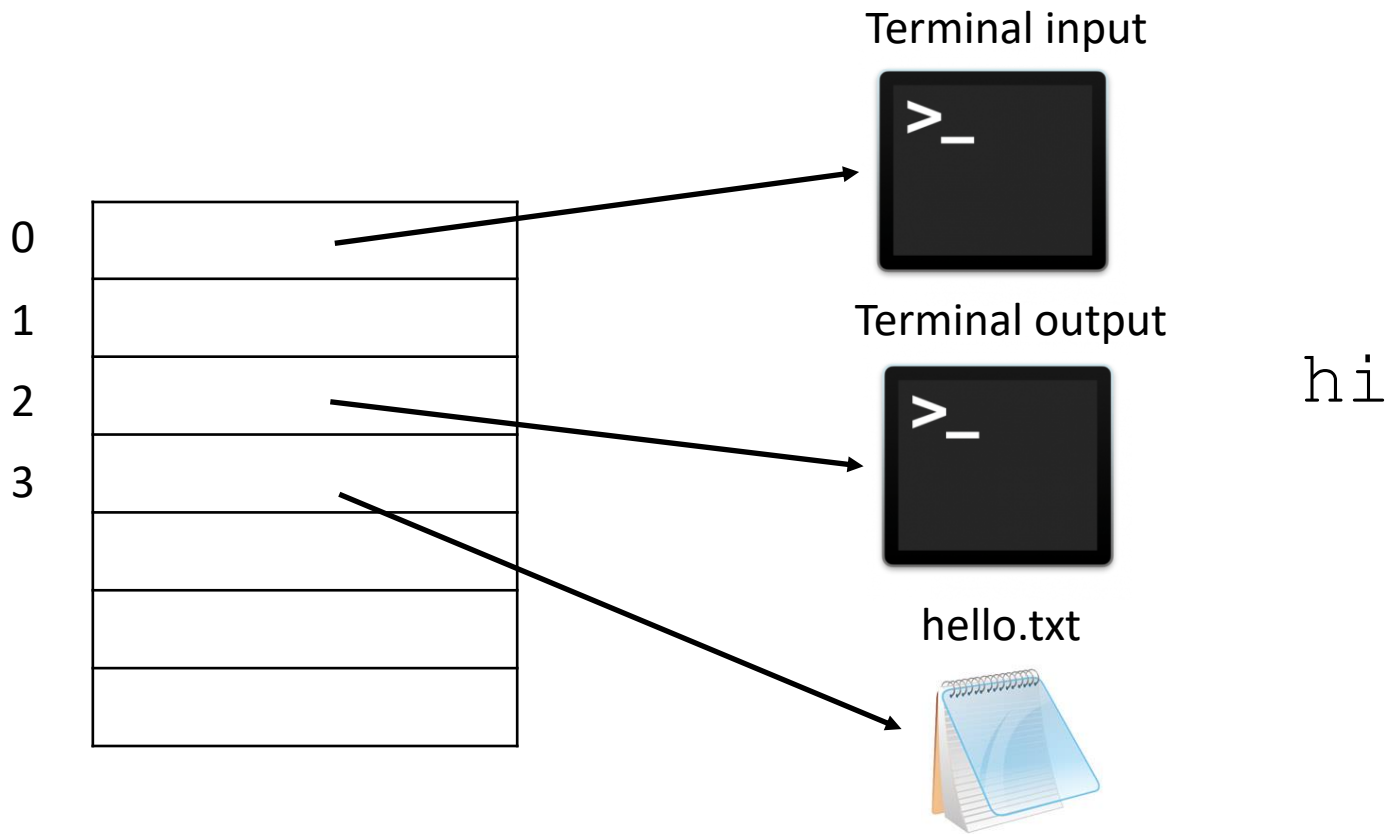
Explanation

```
close(STDOUT_FILENO);
```



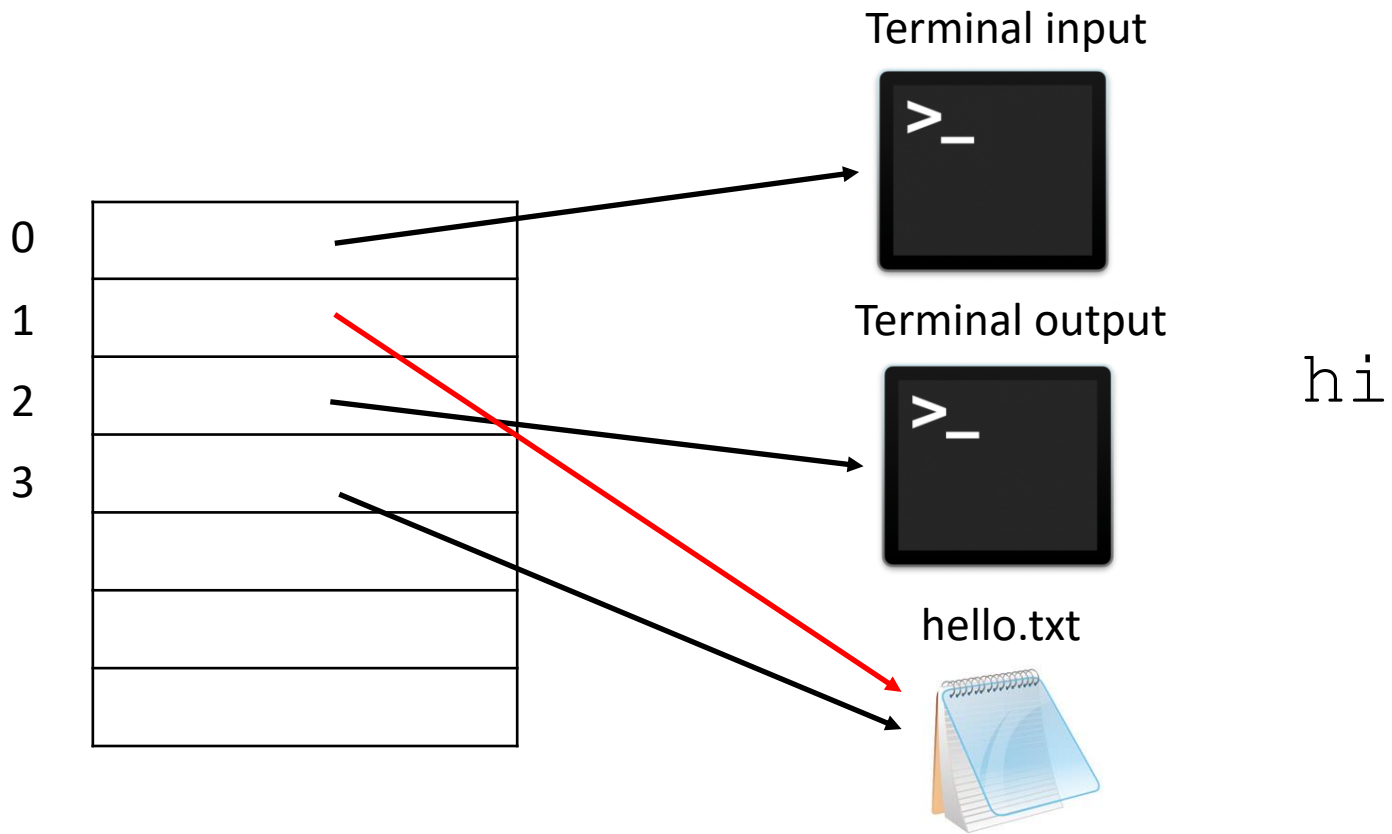
Explanation

```
close(STDOUT_FILENO);
printf("?\n"); // errors! Nothing printed
```



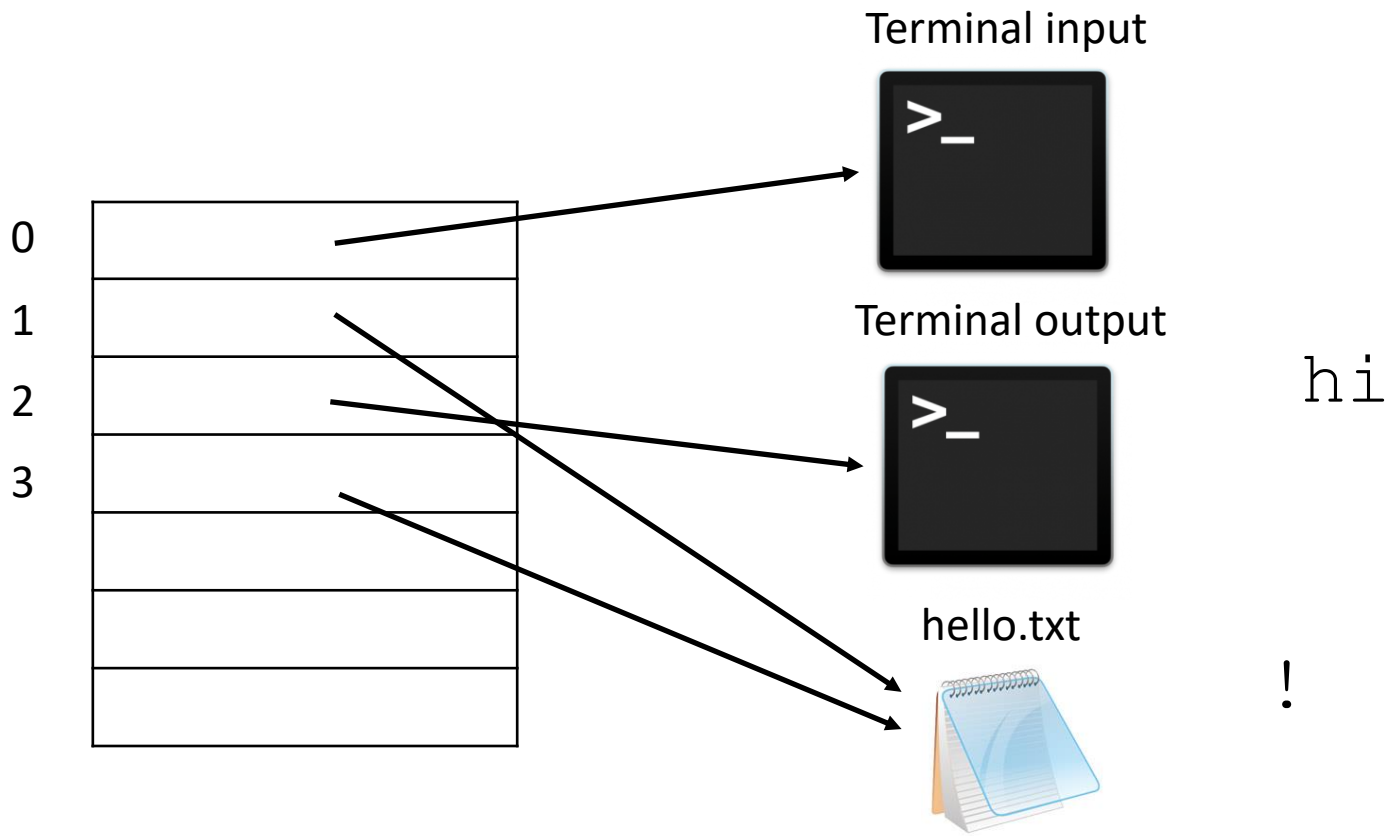
Explanation

```
dup2(fd, STDOUT_FILENO);
```



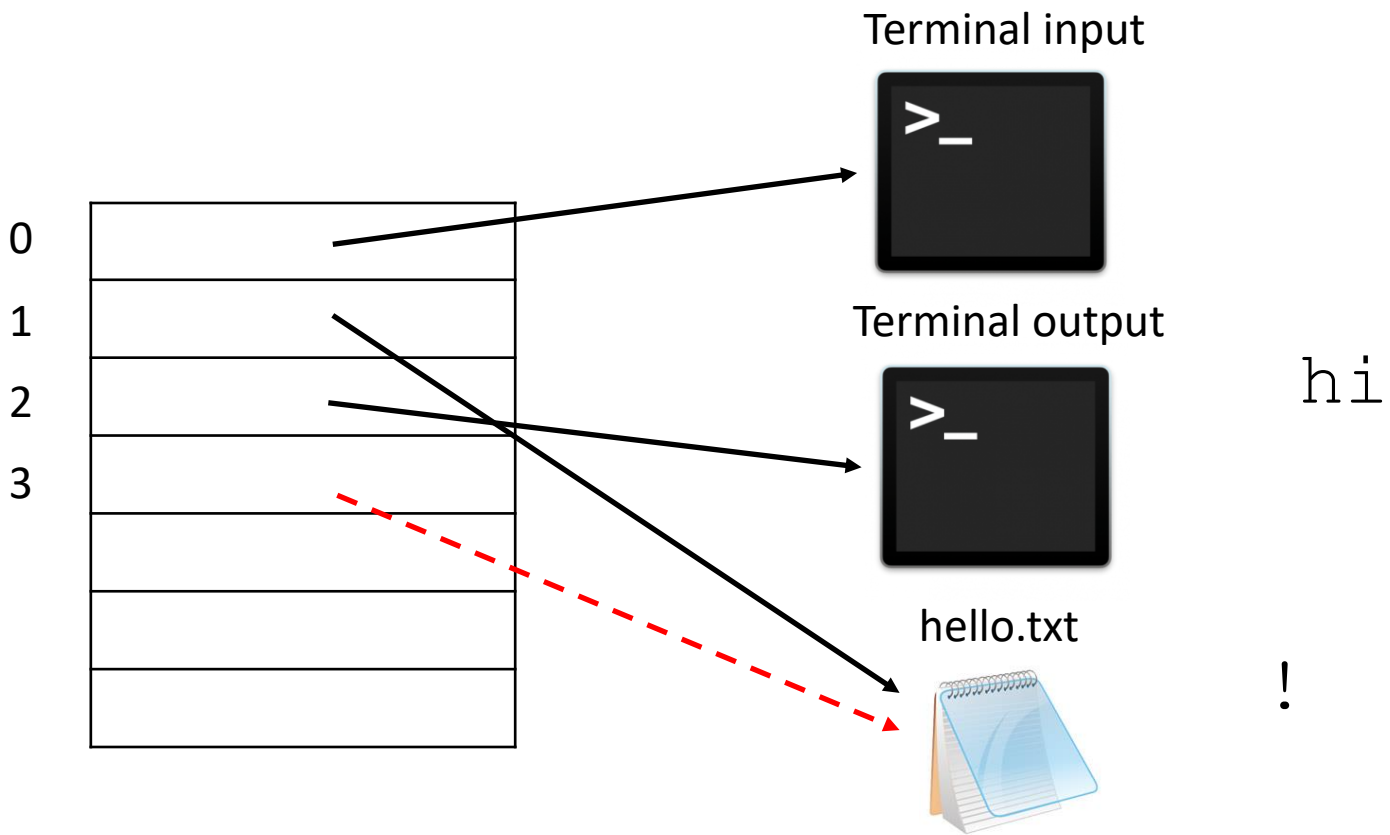
Explanation

```
dup2(fd, STDOUT_FILENO);
printf("!\n");
```



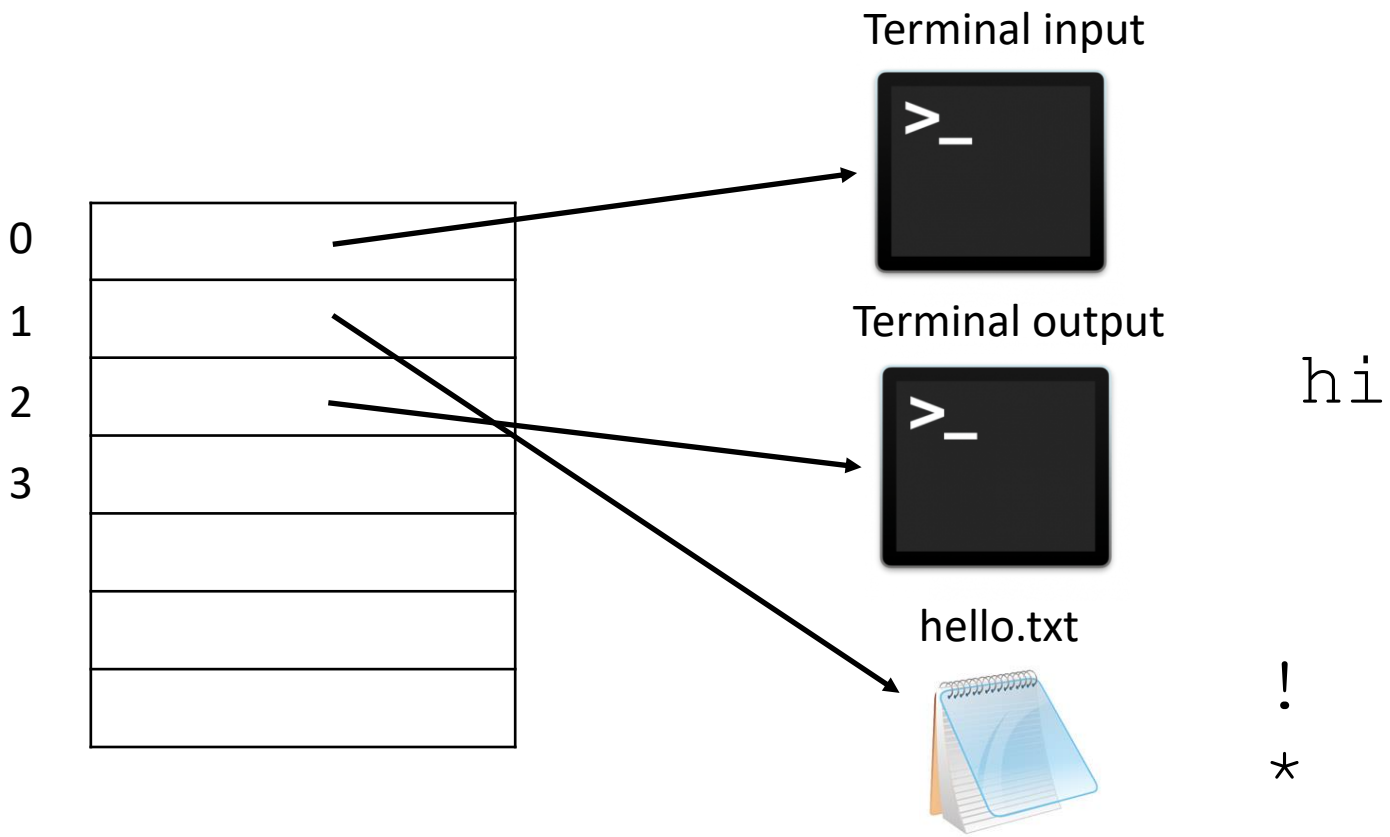
Explanation

```
close(fd);
```



Explanation

```
printf("*\n");
```



Lecture Outline

- C++ & C Interop
- Processes & Fork
- stdin, stdout, stderr & File Descriptors
- **Exec**
- Pipe

exec*()

- Loads in a new program for execution

- PC, SP, registers, and memory are all reset so that the specified program can run

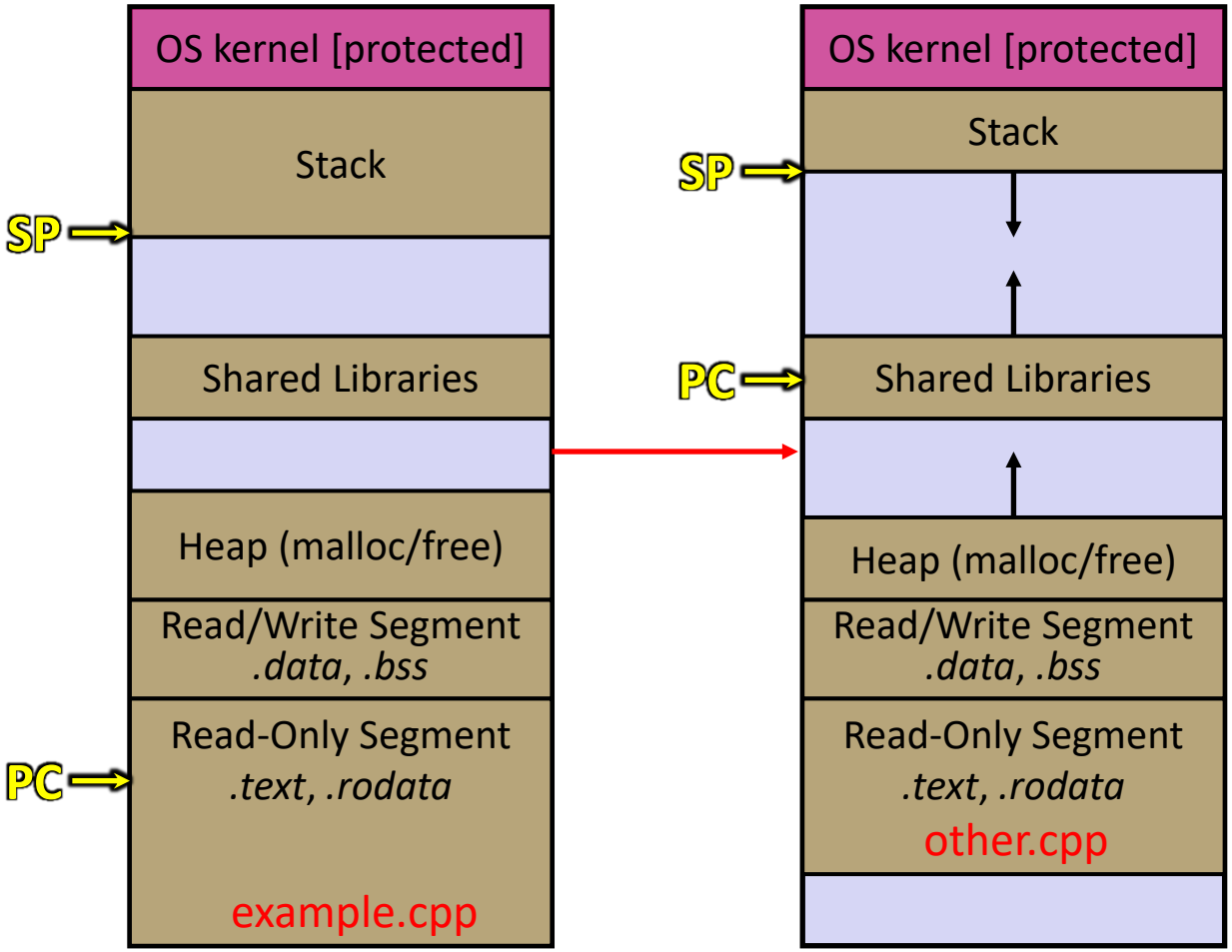
execvp()

```
int execvp(const char *file,
           char* const argv[]);
```

- Duplicates the action of the shell (terminal) in terms of finding the command/program to run
- Argv is an array of **`char*`**, the same kind of argv that is passed to `main()` in a C/C++ program
  - **`argv[0]`** MUST have the same contents as the file parameter
  - **`argv`** must have NULL/nullptr as the last entry of the array
- Returns -1 on error. Does NOT return on success

Exec Visualization

- Exec takes a process and discards or "resets" most of it



NOTE that the following DO change
- The stack
- The heap
- Globals
- Loaded code
- Registers

NOTE that the following do NOT change
- Process ID
- Open files
- The kernel

Exec Demo

- See `exec_example.cpp`
  - Brief code demo to see how exec works
  - What happens when we call exec?
  - What happens if we open some files before exec?
  - What happens if we replace stdout with a file?

- NOTE: When a process exits, then it will close all of its open files by default

Poll Everywhere

pollev.com/tqm

- In each of these, how often is ":)" printed? Assume functions don't fail

```
int main(int argc, char* argv[]) {

  pid_t pid = fork();
  if (pid == 0) {
    // we are the child
    char* argv[] = {"echo",
                    "hello",
                    NULL};
    execvp(argv[0], argv);
  }

  cout << ":)" << endl;

  return EXIT_SUCCESS;
}
```

```
int main(int argc, char* argv[]) {
  char* envp[] = { NULL };

  pid_t pid = fork();
  if (pid == 0) {
    // we are the child
    return EXIT_SUCCESS;
  }

  cout << ":)" << endl;

  return EXIT_SUCCESS;
}
```

Lecture Outline

- C++ & C Interop
- Processes & Fork
- stdin, stdout, stderr & File Descriptors
- Exec
- Pipe

Pipes

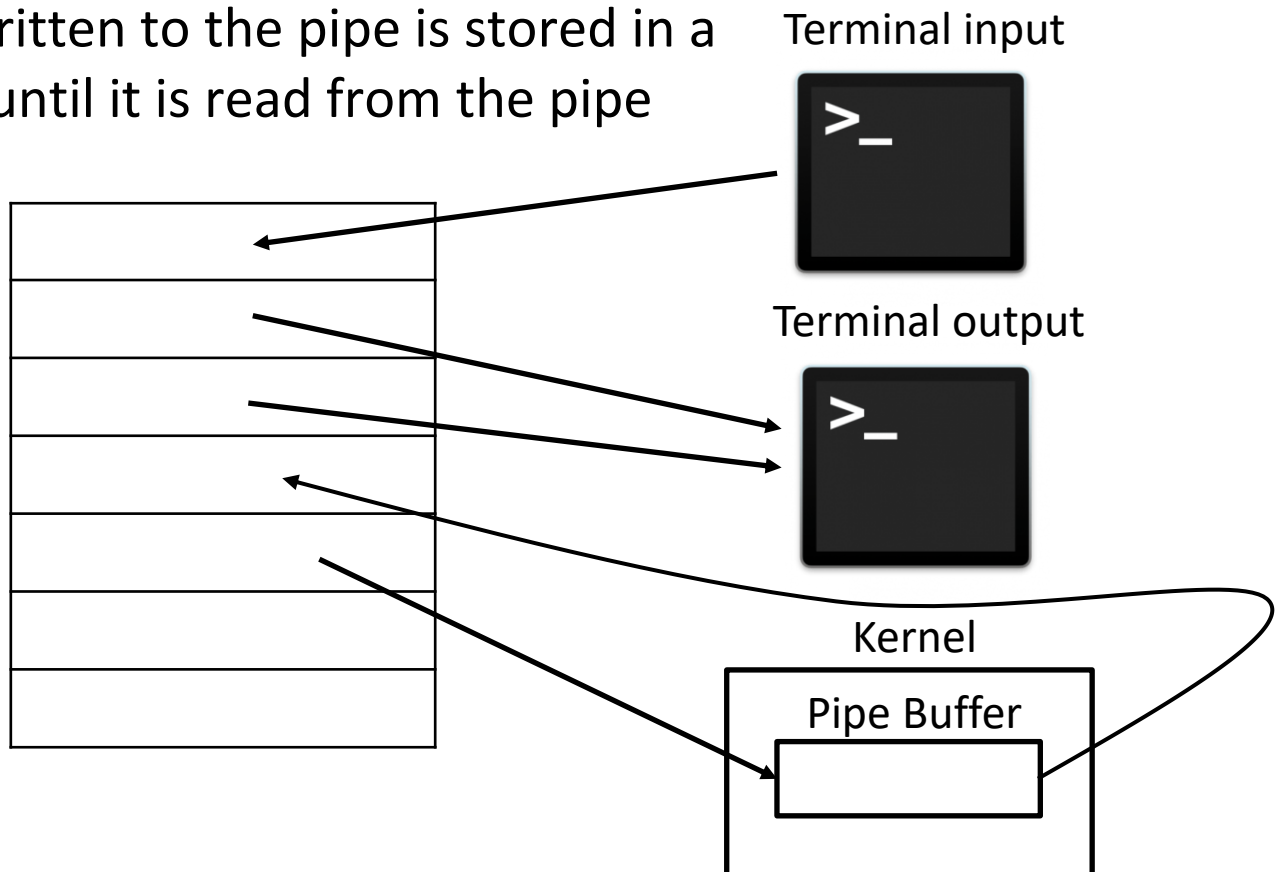
```
int pipe(int pipefd[2]);
```

- 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)

```
12 // writes the string to the specified fd
13 bool wrapped_write(int fd, const string& to_write);
14
15 // reads till eof from specified fd. nullopt on error
16 optional<string> wrapped_read(int fd);
17
18 int main() {
19   int pipe_fds[2];
20   pipe(pipe_fds);
21
22   // child process only exits after this
23   pid_t pid = fork();
24
25   if (pid == 0) {
26     // child process
27
28     // close the end of the pipe that isn't used
29     close(pipe_fds[0]);
30
31     string greeting {"Hello!"};
32     wrapped_write(pipe_fds[1], greeting);
33
34     optional<string> response = wrapped_read(pipe_fds[1]);
35
36     if (response.has_value()) {
37       cout << response.value() << endl;
38     }
39
40     exit(EXIT_SUCCESS);
41   }
42   // parent
```

pipe_unidirect.cpp
on course website

```
42   // parent
43
44   /// close the end of the pipe I won't use
45   close(pipe_fds[1]);
46
47   optional<string> message = wrapped_read(pipe_fds[0]);
48
49   if (message.has_value()) {
50     cout << message.value() << endl;
51   }
52
53   string greeting{"Howdy!"};
54   wrapped_write(pipe_fds[0], greeting);
55
56   int wstatus;
57   waitpid(pid, &wstatus, 0);
58
59   return EXIT_SUCCESS;
60 }
```

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 all!

- More on pipe in next lecture!
- Any questions?