CS110 Lecture 06: Pipes, Signals, and Concurrency

Principles of Computer Systems

Winter 2021

Stanford University

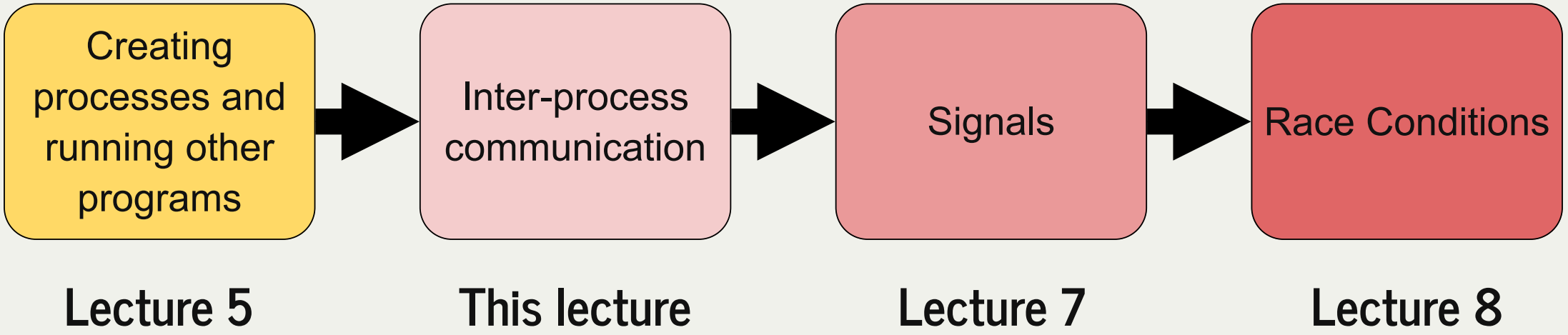
Computer Science Department

Instructors: Chris Gregg and Nick Troccoli

PDF of this presentation

CS110 Topic 2: How can our programs create and interact with other programs?

Learning About Processes



Learning Goals

- Get more practice with using **fork()** and **execvp**
- Learn about **pipe** and **dup2** to create and manipulate file descriptors
- Use pipes to redirect process input and output

Lecture Plan

- Review: **fork()** and **execvp()**
- Running in the background
- Introducing Pipes
- ***Practice:*** Implementing **subprocess**

fork()

- A system call that creates a new *child process*
- The "parent" is the process that creates the other "child" process
- From then on, both processes are running the code after the fork
- The child process is *identical* to the parent, except:
  - it has a new Process ID (PID)
  - for the parent, fork() returns the PID of the child; for the child, fork() returns 0
  - fork() is **called once**, but **returns twice**

```
1 pid_t pidOrZero = fork();
2 // both parent and child run code here onwards
3 printf("This is printed by two processes.\n");
```

waitpid()

A function that a parent can call to wait for its child to exit:

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

- **pid:** the PID of the child to wait on, or -1 to wait on any of our children
- **status:** where to put info about the child's termination (or NULL)
- **options:** optional flags to customize behavior (always 0 for now)

The function returns when the specified **child process** exits.

- the return value is the PID of the child that exited, or -1 on error (e.g. no child to wait on)
- If the child process has already exited, this returns immediately - otherwise, it blocks
- It's important to wait on all children to clean up system resources

execvp()

execvp is a function that lets us run *another program* in the current process.

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

It runs the executable at the specified **path**, *completely cannibalizing the current process.*

- If successful, **execvp never returns** in the calling process
- If unsuccessful, **execvp** returns -1

To run another executable, we must specify the (NULL-terminated) arguments to be passed into its **main** function, via the **argv** parameter.

- For our programs, **path** and **argv[0]** will be the same

execvp has many variants (**execle**, **execlp**, and so forth. Type **man execvp** for more). We rely on **execvp** in CS110.

Revisiting `mysystem`

mysystem is our own version of the built-in function **system**.

- It takes in a terminal command (e.g. "ls -l /usr/class/cs110"), executes it in a separate process, and returns when that process is finished.
  - We can use **fork** to create the child process
  - We can use **execvp** in that child process to execute the terminal command
  - We can use **waitpid** in the parent process to wait for the child to terminate

Revisiting `mysystem`

```
1  static int mysystem(char *command) {
2      pid_t pidOrZero = fork();
3      if (pidOrZero == 0) {
4          char *arguments[] = {"/bin/sh", "-c", command, NULL};
5          execvp(arguments[0], arguments);
6          // If the child gets here, there was an error
7          exitIf(true, kExecFailed, stderr, "execvp failed to invoke this: %s.\n", command);
8      }
9
10     // If we are the parent, wait for the child
11     int status;
12     waitpid(pidOrZero, &status, 0);
13     return WIFEXITED(status) ? WEXITSTATUS(status) : -WTERMSIG(status);
14 }
```

first-shell-soln.c

Revisiting `first-shell`

```
1  int main(int argc, char *argv[]) {
2      char command[kMaxLineLength];
3      while (true) {
4          printf("> ");
5          fgets(command, sizeof(command), stdin);
6
7          // If the user entered Ctl-d, stop
8          if (feof(stdin)) {
9              break;
10         }
11
12         // Remove the \n that fgets puts at the end
13         command[strlen(command) - 1] = '\0';
14
15         int commandReturnCode = mysystem(command);
16         printf("return code = %d\n", commandReturnCode);
17     }
18
19     printf("\n");
20     return 0;
21 }
```

first-shell-soln.c

Our **first-shell** program is a loop in **main** that parses the user input and passes it to **mysystem**.

`first-shell` Takeaways

- A shell is a program that repeats: read command from the user, execute that command
- In order to execute a program and continue running the shell afterwards, we fork off another process and run the program in that process
- We rely on **`fork`**, **`execvp`**, and **`waitpid`** to do this!
- Real shells have more advanced functionality that we will add going forward.
- For your fourth assignment, you'll build on this with your own shell, **`stsh`** ("Stanford shell") with much of the functionality of real Unix shells.

More Shell Functionality

Shells have a variety of supported commands:

- **emacs &** - create an emacs process and run it in the background
- **cat file.txt | uniq | sort** - pipe the output of one command to the input of another
- **uniq < file.txt | sort > list.txt** - make file.txt the input of uniq and output sort to list.txt
- Let's see how we can implement these - but first, a demo.

Lecture Plan

- Review: **fork()** and **execvp()**
- Running in the background
- Introducing Pipes
- ***Practice:*** Implementing **subprocess**

Supporting Background Execution

Let's make an updated version of **mysystem** called **executeCommand**.

Supporting Background Execution

Let's make an updated version of **mysystem** called **executeCommand**.

- Takes an additional parameter **bool inBackground**
  - If **false**, same behavior as **mysystem** (spawn child, execvp, wait for child)
  - If **true**, spawn child, execvp, but *don't wait for child*

Supporting Background Execution

```
1  static void executeCommand(char *command, bool inBackground) {
2      pid_t pidOrZero = fork();
3      if (pidOrZero == 0) {
4          // If we are the child, execute the shell command
5          char *arguments[] = {"/bin/sh", "-c", command, NULL};
6          execvp(arguments[0], arguments);
7          // If the child gets here, there was an error
8          exitIf(true, kExecFailed, stderr, "execvp failed to invoke this: %s.\n", command);
9      }
10
11     // If we are the parent, either wait or return immediately
12     if (inBackground) {
13         printf("%d %s\n", pidOrZero, command);
14     } else {
15         waitpid(pidOrZero, NULL, 0);
16     }
17 }
```

second-shell-start.c

Supporting Background Execution

```
1  static void executeCommand(char *command, bool inBackground) {
2      pid_t pidOrZero = fork();
3      if (pidOrZero == 0) {
4          // If we are the child, execute the shell command
5          char *arguments[] = {"/bin/sh", "-c", command, NULL};
6          execvp(arguments[0], arguments);
7          // If the child gets here, there was an error
8          exitIf(true, kExecFailed, stderr, "execvp failed to invoke this: %s.\n", command);
9      }
10
11     // If we are the parent, either wait or return immediately
12     if (inBackground) {
13         printf("%d %s\n", pidOrZero, command);
14     } else {
15         waitpid(pidOrZero, NULL, 0);
16     }
17 }
```

Line 1: Now, the caller can optionally run the command in the background.

second-shell-start.c

Supporting Background Execution

```
1  static void executeCommand(char *command, bool inBackground) {
2      pid_t pidOrZero = fork();
3      if (pidOrZero == 0) {
4          // If we are the child, execute the shell command
5          char *arguments[] = {"/bin/sh", "-c", command, NULL};
6          execvp(arguments[0], arguments);
7          // If the child gets here, there was an error
8          exitIf(true, kExecFailed, stderr, "execvp failed to invoke this: %s.\n", command);
9      }
10
11     // If we are the parent, either wait or return immediately
12     if (inBackground) {
13         printf("%d %s\n", pidOrZero, command);
14     } else {
15         waitpid(pidOrZero, NULL, 0);
16     }
17 }
```

Lines 11-16: The parent waits on a foreground child, but not a background child.

second-shell-start.c

Supporting Background Execution

```
1  int main(int argc, char *argv[]) {
2      char command[kMaxLineLength];
3      while (true) {
4          printf("> ");
5          fgets(command, sizeof(command), stdin);
6
7          // If the user entered Ctl-d, stop
8          if (feof(stdin)) {
9              break;
10         }
11
12         // Remove the \n that fgets puts at the end
13         command[strlen(command) - 1] = '\0';
14
15         if (strcmp(command, "quit") == 0) break;
16
17         bool isbg = command[strlen(command) - 1] == '&';
18         if (isbg) {
19             command[strlen(command) - 1] = '\0';
20         }
21
22         executeCommand(command, isbg);
23     }
24
25     printf("\n");
26     return 0;
27 }
```

In main, we must add two additional things:

- Check for the "quit" command to exit
- Allow the user to add "&" at the end of a command to run that command in the background

Note that a background child isn't reaped! This is a problem - one we'll learn how to fix soon.

second-shell-start.c

Supporting Background Execution

```
1  int main(int argc, char *argv[]) {
2      char command[kMaxLineLength];
3      while (true) {
4          printf("> ");
5          fgets(command, sizeof(command), stdin);
6
7          // If the user entered Ctl-d, stop
8          if (feof(stdin)) {
9              break;
10         }
11
12         // Remove the \n that fgets puts at the end
13         command[strlen(command) - 1] = '\0';
14
15         if (strcmp(command, "quit") == 0) break;
16
17         bool isbg = command[strlen(command) - 1] == '&';
18         if (isbg) {
19             command[strlen(command) - 1] = '\0';
20         }
21
22         executeCommand(command, isbg);
23     }
24
25     printf("\n");
26     return 0;
27 }
```

In main, we must add two additional things:

- Check for the "quit" command to exit
- Allow the user to add "&" at the end of a command to run that command in the background

Note that a background child isn't reaped! This is a problem - one we'll learn how to fix soon.

second-shell-start.c

Lecture Plan

- Review: **fork()** and **execvp()**
- Running in the background
- Introducing Pipes
- ***Practice:*** Implementing **subprocess**

Lecture Plan

- Review: **fork()** and **execvp()**
- Running in the background
- Introducing Pipes
  - What are pipes?
  - Pipes between processes
  - Redirecting process I/O
- ***Practice:*** Implementing **subprocess**

Is there a way that the parent and child processes can communicate?

Interprocess Communication

- It's useful for a parent process to communicate with its child (and vice versa)
- There are two key ways we will learn to do this: **pipes** and **signals**
  - **Pipes** let two processes send and receive arbitrary data
  - **Signals** let two processes send and receive certain "signals" that indicate something special has happened.

Interprocess Communication

- It's useful for a parent process to communicate with its child (and vice versa)
- There are two key ways we will learn to do this: **pipes** and **signals**
  - **Pipes** let two processes send and receive arbitrary data
  - **Signals** let two processes send and receive certain "signals" that indicate something special has happened.

Pipes


LELAND STANFORD JUNIOR UNIVERSITY
DIE LUFT DER FREIHEIT WEHT
1891

Pipes

- How can we let two processes send arbitrary data back and forth?

Pipes

- How can we let two processes send arbitrary data back and forth?
- A core Unix principle is modeling things as *files*. Could we use a "file"?

Pipes

- How can we let two processes send arbitrary data back and forth?
- A core Unix principle is modeling things as *files*. Could we use a "file"?
- **Idea:** a file that one process could write, and another process could read?

Pipes

- How can we let two processes send arbitrary data back and forth?
- A core Unix principle is modeling things as *files*. Could we use a "file"?
- **Idea:** a file that one process could write, and another process could read?
- **Problem:** we don't want to clutter the filesystem with actual files every time two processes want to communicate.

Pipes

- How can we let two processes send arbitrary data back and forth?
- A core Unix principle is modeling things as *files*. Could we use a "file"?
- **Idea:** a file that one process could write, and another process could read?
- **Problem:** we don't want to clutter the filesystem with actual files every time two processes want to communicate.
- **Solution:** have the operating system set this up for us.
  - It will give us two new file descriptors - one for writing, another for reading.
  - If someone writes data to the write FD, it can be read from the read FD.
  - It's *not actually a physical file on disk* - we are just using files as an abstraction

pipe()

```
int pipe(int fds[]);
```

The **pipe** system call populates the 2-element array **fds** with two file descriptors such that everything *written* to **fds[1]** can be *read* from **fds[0]**. Returns 0 on success, or -1 on error.

pipe()

```
int pipe(int fds[]);
```

The **pipe** system call populates the 2-element array **fds** with two file descriptors such that everything *written* to **`fds[1]`** can be *read* from **`fds[0]`**. Returns 0 on success, or -1 on error.

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      int result = pipe(fds);
5
6      // Write message to pipe (assuming here all bytes written immediately)
7      write(fds[1], kPipeMessage, strlen(kPipeMessage) + 1);
8      close(fds[1]);
9
10     // Read message from pipe
11     char receivedMessage[strlen(kPipeMessage) + 1];
12     read(fds[0], receivedMessage, sizeof(receivedMessage));
13     close(fds[0]);
14     printf("Message read: %s\n", receivedMessage);
15
16     return 0;
17 }
```

Tip: you learn to read before you learn to write (read = fds[0], write = fds[1]).

pipe-demo.c

pipe()

```
int pipe(int fds[]);
```

The **pipe** system call populates the 2-element array **fds** with two file descriptors such that everything *written* to **`fds[1]`** can be *read* from **`fds[0]`**. Returns 0 on success, or -1 on error.

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      int result = pipe(fds);
5
6      // Write message to pipe (assuming here all bytes written immediately)
7      write(fds[1], kPipeMessage, strlen(kPipeMessage) + 1);
8      close(fds[1]);
9
10     // Read message from pipe
11     char receivedMessage[strlen(kPipeMessage) + 1];
12     read(fds[0], receivedMessage, sizeof(receivedMessage));
13     close(fds[0]);
14     printf("Message read: %s\n", receivedMessage);
15
16     return 0;
17 }
```

Tip: you learn to read before you learn to write (read = fds[0], write = fds[1]).

pipe-demo.c

pipe()

```
int pipe(int fds[]);
```

The **pipe** system call populates the 2-element array **fds** with two file descriptors such that everything *written* to **`fds[1]`** can be *read* from **`fds[0]`**. Returns 0 on success, or -1 on error.

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      int result = pipe(fds);
5
6      // Write message to pipe (assuming here all bytes written immediately)
7      write(fds[1], kPipeMessage, strlen(kPipeMessage) + 1);
8      close(fds[1]);
9
10     // Read message from pipe
11     char receivedMessage[strlen(kPipeMessage) + 1];
12     read(fds[0], receivedMessage, sizeof(receivedMessage));
13     close(fds[0]);
14     printf("Message read: %s\n", receivedMessage);
15
16     return 0;
17 }
```

Tip: you learn to read before you learn to write (read = fds[0], write = fds[1]).

pipe-demo.c

pipe()

```
int pipe(int fds[]);
```

The **pipe** system call populates the 2-element array **fds** with two file descriptors such that everything *written* to **`fds[1]`** can be *read* from **`fds[0]`**. Returns 0 on success, or -1 on error.

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      int result = pipe(fds);
5
6      // Write message to pipe (assuming here all bytes written immediately)
7      write(fds[1], kPipeMessage, strlen(kPipeMessage) + 1);
8      close(fds[1]);
9
10     // Read message from pipe
11     char receivedMessage[strlen(kPipeMessage) + 1];
12     read(fds[0], receivedMessage, sizeof(receivedMessage));
13     close(fds[0]);
14     printf("Message read: %s\n", receivedMessage);
15
16     return 0;
17 }
```

Tip: you learn to read before you learn to write (read = fds[0], write = fds[1]).

pipe-demo.c

pipe()

```
int pipe(int fds[]);
```

The **pipe** system call populates the 2-element array **fds** with two file descriptors such that everything *written* to **`fds[1]`** can be *read* from **`fds[0]`**. Returns 0 on success, or -1 on error.

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      int result = pipe(fds);
5
6      // Write message to pipe (assuming here all bytes written immediately)
7      write(fds[1], kPipeMessage, strlen(kPipeMessage) + 1);
8      close(fds[1]);
9
10     // Read message from pipe
11     char receivedMessage[strlen(kPipeMessage) + 1];
12     read(fds[0], receivedMessage, sizeof(receivedMessage));
13     close(fds[0]);
14     printf("Message read: %s\n", receivedMessage);
15
16     return 0;
17 }
```

Tip: you learn to read before you learn to write (read = fds[0], write = fds[1]).

pipe-demo.c

pipe()

```
int pipe(int fds[]);
```

The **pipe** system call populates the 2-element array **fds** with two file descriptors such that everything *written* to **`fds[1]`** can be *read* from **`fds[0]`**. Returns 0 on success, or -1 on error.

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      int result = pipe(fds);
5
6      // Write message to pipe (assuming here all bytes written immediately)
7      write(fds[1], kPipeMessage, strlen(kPipeMessage) + 1);
8      close(fds[1]);
9
10     // Read message from pipe
11     char receivedMessage[strlen(kPipeMessage) + 1];
12     read(fds[0], receivedMessage, sizeof(receivedMessage));
13     close(fds[0]);
14     printf("Message read: %s\n", receivedMessage);
15
16     return 0;
17 }
```

Tip: you learn to read before you learn to write (read = fds[0], write = fds[1]).

pipe-demo.c

pipe()

```
int pipe(int fds[]);
```

The **pipe** system call populates the 2-element array **fds** with two file descriptors such that everything *written* to **`fds[1]`** can be *read* from **`fds[0]`**. Returns 0 on success, or -1 on error.

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      int result = pipe(fds);
5
6      // Write message to pipe (assuming here all bytes written immediately)
7      write(fds[1], kPipeMessage, strlen(kPipeMessage) + 1);
8      close(fds[1]);
9
10     // Read message from pipe
11     char receivedMessage[strlen(kPipeMessage) + 1];
12     read(fds[0], receivedMessage, sizeof(receivedMessage));
13     close(fds[0]);
14     printf("Message read: %s\n", receivedMessage);
15
16     return 0;
17 }
```

Tip: you learn to read before you learn to write (read = fds[0], write = fds[1]).

pipe-demo.c

pipe()

```
int pipe(int fds[]);
```

The **pipe** system call populates the 2-element array **fds** with two file descriptors such that everything *written* to **`fds[1]`** can be *read* from **`fds[0]`**. Returns 0 on success, or -1 on error.

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      int result = pipe(fds);
5
6      // Write message to pipe (assuming here all bytes written immediately)
7      write(fds[1], kPipeMessage, strlen(kPipeMessage) + 1);
8      close(fds[1]);
9
10     // Read message from pipe
11     char receivedMessage[strlen(kPipeMessage) + 1];
12     read(fds[0], receivedMessage, sizeof(receivedMessage));
13     close(fds[0]);
14     printf("Message read: %s\n", receivedMessage);
15
16     return 0;
17 }
```

Tip: you learn to read before you learn to write (read = fds[0], write = fds[1]).

pipe-demo.c

pipe()

```
int pipe(int fds[]);
```

The **pipe** system call populates the 2-element array **fds** with two file descriptors such that everything *written* to **fds[1]** can be *read* from **fds[0]**. Returns 0 on success, or -1 on error.

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      int result = pipe(fds);
5
6      // Write message to pipe (assuming here all bytes written immediately)
7      write(fds[1], kPipeMessage, strlen(kPipeMessage) + 1);
8      close(fds[1]);
9
10     // Read message from pipe
11     char receivedMessage[strlen(kPipeMessage) + 1];
12     read(fds[0], receivedMessage, sizeof(receivedMessage));
13     close(fds[0]);
14     printf("Message read: %s\n", receivedMessage);
15
16     return 0;
17 }
```

```
1 $ ./pipe-demo
2 Message read: Hello, this message is coming through a pipe.
```

Tip: you learn to read before you learn to write (read = fds[0], write = fds[1]).

pipe-demo.c

Lecture Plan

- Review: **fork()** and **execvp()**
- Running in the background
- Introducing Pipes
  - What are pipes?
  - Pipes between processes
  - Redirecting process I/O
- ***Practice:*** Implementing **subprocess**

pipe()

```
int pipe(int fds[]);
```

The **pipe** system call populates the 2-element array **fds** with two file descriptors such that everything *written* to **fds[1]** can be *read* from **fds[0]**. Returns 0 on success, or -1 on error.

pipe can allow processes to communicate!

pipe()

```
int pipe(int fds[]);
```

The **pipe** system call populates the 2-element array **fds** with two file descriptors such that everything *written* to **fds[1]** can be *read* from **fds[0]**. Returns 0 on success, or -1 on error.

pipe can allow processes to communicate!

- The parent's file descriptor table is replicated in the child - both have pipe access

pipe()

```
int pipe(int fds[]);
```

The **pipe** system call populates the 2-element array **fds** with two file descriptors such that everything *written* to **fds[1]** can be *read* from **fds[0]**. Returns 0 on success, or -1 on error.

pipe can allow processes to communicate!

- The parent's file descriptor table is replicated in the child - both have pipe access
- E.g. the parent can write to the "write" end and the child can read from the "read" end

pipe()

```
int pipe(int fds[]);
```

The **pipe** system call populates the 2-element array **fds** with two file descriptors such that everything *written* to **fds[1]** can be *read* from **fds[0]**. Returns 0 on success, or -1 on error.

pipe can allow processes to communicate!

- The parent's file descriptor table is replicated in the child - both have pipe access
- E.g. the parent can write to the "write" end and the child can read from the "read" end
- Because they're file descriptors, there's no global name for the pipe (another process can't "connect" to the pipe).

pipe()

```
int pipe(int fds[]);
```

The **pipe** system call populates the 2-element array **fds** with two file descriptors such that everything *written* to **fds[1]** can be *read* from **fds[0]**. Returns 0 on success, or -1 on error.

pipe can allow processes to communicate!

- The parent's file descriptor table is replicated in the child - both have pipe access
- E.g. the parent can write to the "write" end and the child can read from the "read" end
- Because they're file descriptors, there's no global name for the pipe (another process can't "connect" to the pipe).
- Each pipe is uni-directional (one end is read, the other write)

pipe()

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      pipe(fds);
5      size_t bytesSent = strlen(kPipeMessage) + 1;
6
7      pid_t pidOrZero = fork();
8      if (pidOrZero == 0) {
9          // In the child, we only read from the pipe
10         close(fds[1]);
11         char buffer[bytesSent];
12         read(fds[0], buffer, sizeof(buffer));
13         close(fds[0]);
14         printf("Message from parent: %s\n", buffer);
15         return 0;
16     }
17
18     // In the parent, we only write to the pipe (assume everything is written)
19     close(fds[0]);
20     write(fds[1], kPipeMessage, bytesSent);
21     close(fds[1]);
22     waitpid(pidOrZero, NULL, 0);
23     return 0;
24 }
```

Here's an example program showing how pipe works across processes (full program link at bottom).

parent-child-pipe.c

pipe()

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      pipe(fds);
5      size_t bytesSent = strlen(kPipeMessage) + 1;
6
7      pid_t pidOrZero = fork();
8      if (pidOrZero == 0) {
9          // In the child, we only read from the pipe
10         close(fds[1]);
11         char buffer[bytesSent];
12         read(fds[0], buffer, sizeof(buffer));
13         close(fds[0]);
14         printf("Message from parent: %s\n", buffer);
15         return 0;
16     }
17
18     // In the parent, we only write to the pipe (assume everything is written)
19     close(fds[0]);
20     write(fds[1], kPipeMessage, bytesSent);
21     close(fds[1]);
22     waitpid(pidOrZero, NULL, 0);
23     return 0;
24 }
```

Make a pipe just like before.

parent-child-pipe.c

pipe()

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      pipe(fds);
5      size_t bytesSent = strlen(kPipeMessage) + 1;
6
7      pid_t pidOrZero = fork();
8      if (pidOrZero == 0) {
9          // In the child, we only read from the pipe
10         close(fds[1]);
11         char buffer[bytesSent];
12         read(fds[0], buffer, sizeof(buffer));
13         close(fds[0]);
14         printf("Message from parent: %s\n", buffer);
15         return 0;
16     }
17
18     // In the parent, we only write to the pipe (assume everything is written)
19     close(fds[0]);
20     write(fds[1], kPipeMessage, bytesSent);
21     close(fds[1]);
22     waitpid(pidOrZero, NULL, 0);
23     return 0;
24 }
```

parent-child-pipe.c

The parent must close all its open FDs. It never uses the Read FD so we can close it here.

pipe()

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      pipe(fds);
5      size_t bytesSent = strlen(kPipeMessage) + 1;
6
7      pid_t pidOrZero = fork();
8      if (pidOrZero == 0) {
9          // In the child, we only read from the pipe
10         close(fds[1]);
11         char buffer[bytesSent];
12         read(fds[0], buffer, sizeof(buffer));
13         close(fds[0]);
14         printf("Message from parent: %s\n", buffer);
15         return 0;
16     }
17
18     // In the parent, we only write to the pipe (assume everything is written)
19     close(fds[0]);
20     write(fds[1], kPipeMessage, bytesSent);
21     close(fds[1]);
22     waitpid(pidOrZero, NULL, 0);
23     return 0;
24 }
```

parent-child-pipe.c

Write to the Write FD to send a message to the child.

pipe()

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      pipe(fds);
5      size_t bytesSent = strlen(kPipeMessage) + 1;
6
7      pid_t pidOrZero = fork();
8      if (pidOrZero == 0) {
9          // In the child, we only read from the pipe
10         close(fds[1]);
11         char buffer[bytesSent];
12         read(fds[0], buffer, sizeof(buffer));
13         close(fds[0]);
14         printf("Message from parent: %s\n", buffer);
15         return 0;
16     }
17
18     // In the parent, we only write to the pipe (assume everything is written)
19     close(fds[0]);
20     write(fds[1], kPipeMessage, bytesSent);
21     close(fds[1]);
22     waitpid(pidOrZero, NULL, 0);
23     return 0;
24 }
```

parent-child-pipe.c

We are now done with the Write FD so we can close it here.

pipe()

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      pipe(fds);
5      size_t bytesSent = strlen(kPipeMessage) + 1;
6
7      pid_t pidOrZero = fork();
8      if (pidOrZero == 0) {
9          // In the child, we only read from the pipe
10         close(fds[1]);
11         char buffer[bytesSent];
12         read(fds[0], buffer, sizeof(buffer));
13         close(fds[0]);
14         printf("Message from parent: %s\n", buffer);
15         return 0;
16     }
17
18     // In the parent, we only write to the pipe (assume everything is written)
19     close(fds[0]);
20     write(fds[1], kPipeMessage, bytesSent);
21     close(fds[1]);
22     waitpid(pidOrZero, NULL, 0);
23     return 0;
24 }
```

parent-child-pipe.c

We wait for the child to terminate.

pipe()

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      pipe(fds);
5      size_t bytesSent = strlen(kPipeMessage) + 1;
6
7      pid_t pidOrZero = fork();
8      if (pidOrZero == 0) {
9          // In the child, we only read from the pipe
10         close(fds[1]);
11         char buffer[bytesSent];
12         read(fds[0], buffer, sizeof(buffer));
13         close(fds[0]);
14         printf("Message from parent: %s\n", buffer);
15         return 0;
16     }
17
18     // In the parent, we only write to the pipe (assume everything is written)
19     close(fds[0]);
20     write(fds[1], kPipeMessage, bytesSent);
21     close(fds[1]);
22     waitpid(pidOrZero, NULL, 0);
23     return 0;
24 }
```

Key Idea: when we call **fork**, the child gets a copy of the parent's file descriptor table. Any open FDs in the parent at the time **fork** is called must be closed *in both the parent and the child*.

parent-child-pipe.c

pipe()

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      pipe(fds);
5      size_t bytesSent = strlen(kPipeMessage) + 1;
6
7      pid_t pidOrZero = fork();
8      if (pidOrZero == 0) {
9          // In the child, we only read from the pipe
10         close(fds[1]);
11         char buffer[bytesSent];
12         read(fds[0], buffer, sizeof(buffer));
13         close(fds[0]);
14         printf("Message from parent: %s\n", buffer);
15         return 0;
16     }
17
18     // In the parent, we only write to the pipe (assume everything is written)
19     close(fds[0]);
20     write(fds[1], kPipeMessage, bytesSent);
21     close(fds[1]);
22     waitpid(pidOrZero, NULL, 0);
23     return 0;
24 }
```

More specifically, this duplication means the child's file descriptor table entries point to the same open file table entries as the parent. Thus, the open file table entries for the two pipe FDs both have reference counts of 2.

parent-child-pipe.c

pipe()

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      pipe(fds);
5      size_t bytesSent = strlen(kPipeMessage) + 1;
6
7      pid_t pidOrZero = fork();
8      if (pidOrZero == 0) {
9          // In the child, we only read from the pipe
10         close(fds[1]);
11         char buffer[bytesSent];
12         read(fds[0], buffer, sizeof(buffer));
13         close(fds[0]);
14         printf("Message from parent: %s\n", buffer);
15         return 0;
16     }
17
18     // In the parent, we only write to the pipe (assume everything is written)
19     close(fds[0]);
20     write(fds[1], kPipeMessage, bytesSent);
21     close(fds[1]);
22     waitpid(pidOrZero, NULL, 0);
23     return 0;
24 }
```

The child must close all its open FDs. It never uses the Write FD so we can close it here.

parent-child-pipe.c

pipe()

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      pipe(fds);
5      size_t bytesSent = strlen(kPipeMessage) + 1;
6
7      pid_t pidOrZero = fork();
8      if (pidOrZero == 0) {
9          // In the child, we only read from the pipe
10         close(fds[1]);
11         char buffer[bytesSent];
12         read(fds[0], buffer, sizeof(buffer));
13         close(fds[0]);
14         printf("Message from parent: %s\n", buffer);
15         return 0;
16     }
17
18     // In the parent, we only write to the pipe (assume everything is written)
19     close(fds[0]);
20     write(fds[1], kPipeMessage, bytesSent);
21     close(fds[1]);
22     waitpid(pidOrZero, NULL, 0);
23     return 0;
24 }
```

Read from the Read FD to read the message from the parent.

parent-child-pipe.c

pipe()

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      pipe(fds);
5      size_t bytesSent = strlen(kPipeMessage) + 1;
6
7      pid_t pidOrZero = fork();
8      if (pidOrZero == 0) {
9          // In the child, we only read from the pipe
10         close(fds[1]);
11         char buffer[bytesSent];
12         read(fds[0], buffer, sizeof(buffer));
13         close(fds[0]);
14         printf("Message from parent: %s\n", buffer);
15         return 0;
16     }
17
18     // In the parent, we only write to the pipe (assume everything is written)
19     close(fds[0]);
20     write(fds[1], kPipeMessage, bytesSent);
21     close(fds[1]);
22     waitpid(pidOrZero, NULL, 0);
23     return 0;
24 }
```

We are now done with the Read FD so we can close it here. Also print the received message.

parent-child-pipe.c

pipe()

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      pipe(fds);
5      size_t bytesSent = strlen(kPipeMessage) + 1;
6
7      pid_t pidOrZero = fork();
8      if (pidOrZero == 0) {
9          // In the child, we only read from the pipe
10         close(fds[1]);
11         char buffer[bytesSent];
12         read(fds[0], buffer, sizeof(buffer));
13         close(fds[0]);
14         printf("Message from parent: %s\n", buffer);
15         return 0;
16     }
17
18     // In the parent, we only write to the pipe (assume everything is written)
19     close(fds[0]);
20     write(fds[1], kPipeMessage, bytesSent);
21     close(fds[1]);
22     waitpid(pidOrZero, NULL, 0);
23     return 0;
24 }
```

Key Idea: the child gets a copy of the parent's file descriptor table. Any open FDs in the parent at the time **fork** is called must be closed *in both the parent and the child*.

parent-child-pipe.c

pipe()

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      pipe(fds);
5      size_t bytesSent = strlen(kPipeMessage) + 1;
6
7      pid_t pidOrZero = fork();
8      if (pidOrZero == 0) {
9          // In the child, we only read from the pipe
10         close(fds[1]);
11         char buffer[bytesSent];
12         read(fds[0], buffer, sizeof(buffer));
13         close(fds[0]);
14         printf("Message from parent: %s\n", buffer);
15         return 0;
16     }
17
18     // In the parent, we only write to the pipe (assume everything is written)
19     close(fds[0]);
20     write(fds[1], kPipeMessage, bytesSent);
21     close(fds[1]);
22     waitpid(pidOrZero, NULL, 0);
23     return 0;
24 }
```

parent-child-pipe.c

Key Idea: the child gets a copy of the parent's file descriptor table. Any open FDs in the parent at the time **fork** is called must be closed *in both the parent and the child*.

Here, right before the **fork** call, the parent has 2 open file descriptors (besides 0-2): the pipe Read FD and Write FD.

pipe()

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      pipe(fds);
5      size_t bytesSent = strlen(kPipeMessage) + 1;
6
7      pid_t pidOrZero = fork();
8      if (pidOrZero == 0) {
9          // In the child, we only read from the pipe
10         close(fds[1]);
11         char buffer[bytesSent];
12         read(fds[0], buffer, sizeof(buffer));
13         close(fds[0]);
14         printf("Message from parent: %s\n", buffer);
15         return 0;
16     }
17
18     // In the parent, we only write to the pipe (assume everything is written)
19     close(fds[0]);
20     write(fds[1], kPipeMessage, bytesSent);
21     close(fds[1]);
22     waitpid(pidOrZero, NULL, 0);
23     return 0;
24 }
```

parent-child-pipe.c

Key Idea: the child gets a copy of the parent's file descriptor table. Any open FDs in the parent at the time **fork** is called must be closed *in both the parent and the child*.

Therefore, when the child is spawned, it *also* has the same 2 open file descriptors (besides 0-2): the pipe Read FD and Write FD.

pipe()

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      pipe(fds);
5      size_t bytesSent = strlen(kPipeMessage) + 1;
6
7      pid_t pidOrZero = fork();
8      if (pidOrZero == 0) {
9          // In the child, we only read from the pipe
10         close(fds[1]);
11         char buffer[bytesSent];
12         read(fds[0], buffer, sizeof(buffer));
13         close(fds[0]);
14         printf("Message from parent: %s\n", buffer);
15         return 0;
16     }
17
18     // In the parent, we only write to the pipe (assume everything is written)
19     close(fds[0]);
20     write(fds[1], kPipeMessage, bytesSent);
21     close(fds[1]);
22     waitpid(pidOrZero, NULL, 0);
23     return 0;
24 }
```

parent-child-pipe.c

Key Idea: the child gets a copy of the parent's file descriptor table. Any open FDs in the parent at the time **fork** is called must be closed *in both the parent and the child*.

We should close FDs when we are done with them. The **parent** closes them here.

pipe()

```
1  static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
2  int main(int argc, char *argv[]) {
3      int fds[2];
4      pipe(fds);
5      size_t bytesSent = strlen(kPipeMessage) + 1;
6
7      pid_t pidOrZero = fork();
8      if (pidOrZero == 0) {
9          // In the child, we only read from the pipe
10         close(fds[1]);
11         char buffer[bytesSent];
12         read(fds[0], buffer, sizeof(buffer));
13         close(fds[0]);
14         printf("Message from parent: %s\n", buffer);
15         return 0;
16     }
17
18     // In the parent, we only write to the pipe (assume everything is written)
19     close(fds[0]);
20     write(fds[1], kPipeMessage, bytesSent);
21     close(fds[1]);
22     waitpid(pidOrZero, NULL, 0);
23     return 0;
24 }
```

parent-child-pipe.c

Key Idea: the child gets a copy of the parent's file descriptor table. Any open FDs in the parent at the time **fork** is called must be closed *in both the parent and the child*.

We should close FDs when we are done with them. The **child** closes them here.

Trying Out Pipes

https://cplayground.com/?p=eagle-fish-mouse

Pipes

This method of communication between processes relies on the fact that file descriptors are duplicated when forking.

- each process has its own copy of both file descriptors for the pipe
- both processes could read or write to the pipe if they wanted.
- each process must therefore close both file descriptors for the pipe when finished

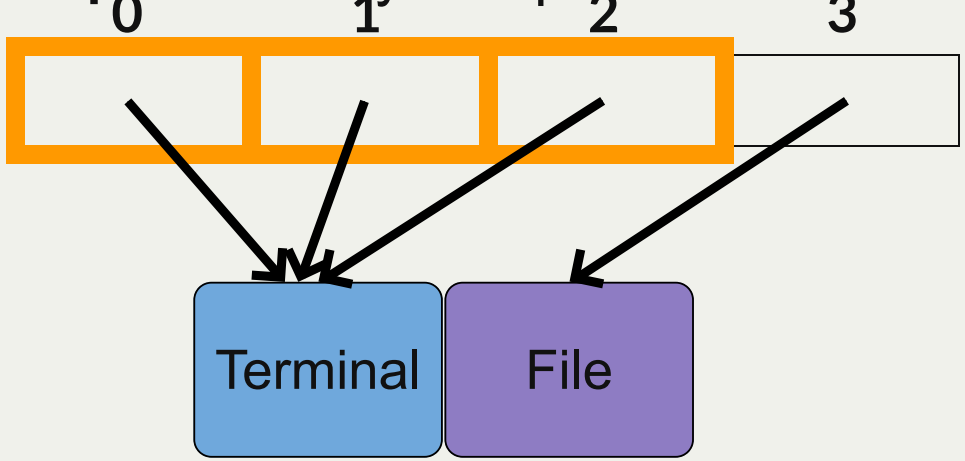
This is the core idea behind how a shell can support piping between processes (e.g. **cat file.txt | uniq | sort**). Let's see how this works in a shell.

Lecture Plan

- Review: **fork()** and **execvp()**
- Running in the background
- Introducing Pipes
  - What are pipes?
  - Pipes between processes
  - Redirecting process I/O
- ***Practice:*** Implementing **subprocess**

Redirecting Process I/O

- Each process has the special file descriptors STDIN (0), STDOUT (1) and STDERR (2)
- Processes assume these indexes are for these methods of communication (e.g. **printf** always outputs to file descriptor 1, STDOUT).

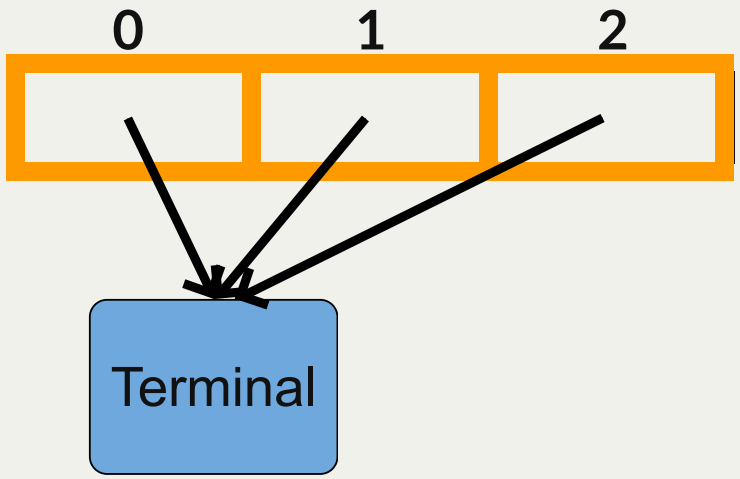


Idea: what happens if we change FD 1 to point somewhere else?

Redirecting Process I/O

Idea: what happens if we change FD 1 to point somewhere else?

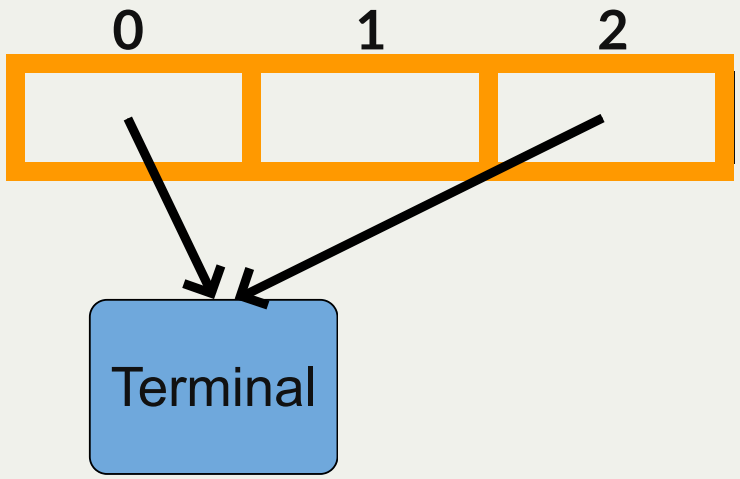
```
 1 int main() {
 2     printf("This will print to the terminal\n");
 3     close(STDOUT_FILENO);
 4
 5     // fd will always be 1
 6     int fd = open("myfile.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
 7
 8     printf("This will print to myfile.txt!\n");
 9     close(fd);
10     return 0;
11 }
```



Redirecting Process I/O

Idea: what happens if we change FD 1 to point somewhere else?

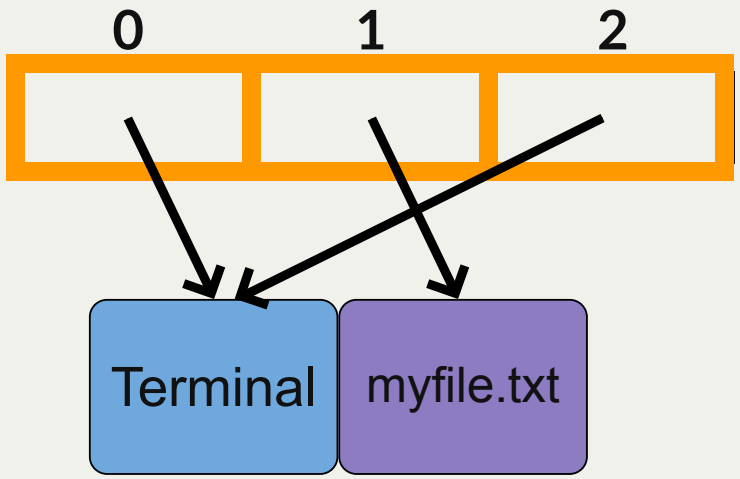
```
 1 int main() {
 2     printf("This will print to the terminal\n");
 3     close(STDOUT_FILENO);
 4
 5     // fd will always be 1
 6     int fd = open("myfile.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
 7
 8     printf("This will print to myfile.txt!\n");
 9     close(fd);
10     return 0;
11 }
```



Redirecting Process I/O

Idea: what happens if we change FD 1 to point somewhere else?

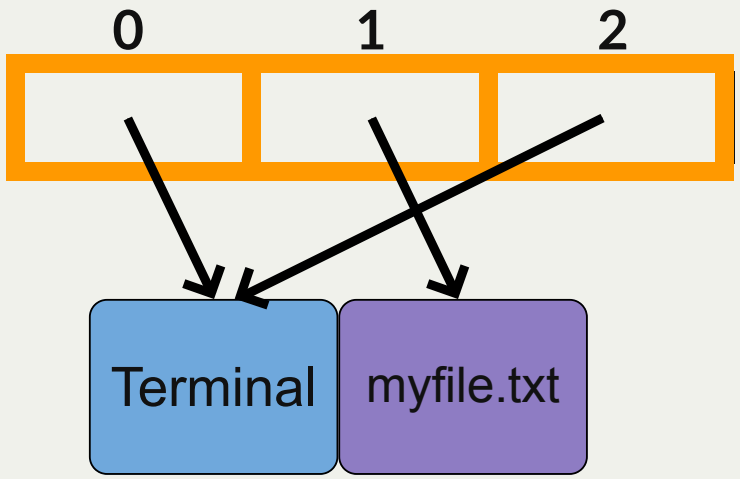
```
 1 int main() {
 2     printf("This will print to the terminal\n");
 3     close(STDOUT_FILENO);
 4
 5     // fd will always be 1
 6     int fd = open("myfile.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
 7
 8     printf("This will print to myfile.txt!\n");
 9     close(fd);
10     return 0;
11 }
```



Redirecting Process I/O

Idea: what happens if we change FD 1 to point somewhere else?

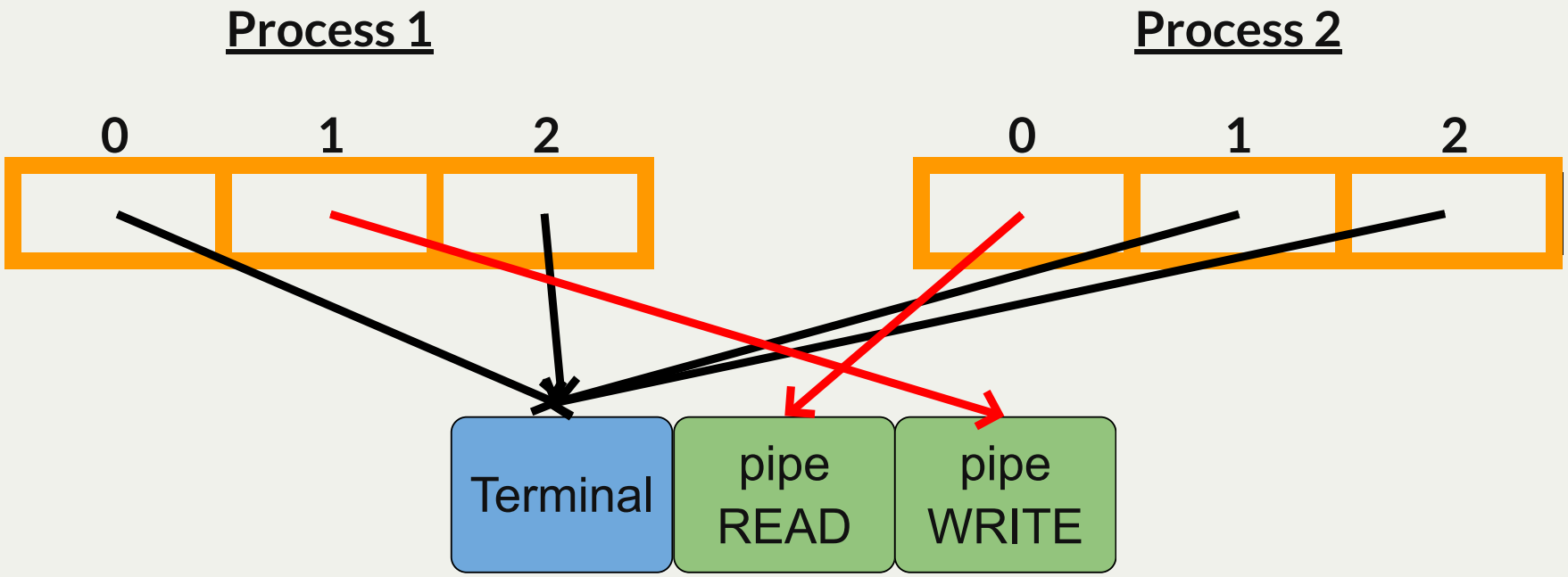
```
1  int main() {
2      printf("This will print to the terminal\n");
3      close(STDOUT_FILENO);
4
5      // fd will always be 1
6      int fd = open("myfile.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
7
8      printf("This will print to myfile.txt!\n");
9      close(fd);
10     return 0;
11 }
```



Redirecting Process I/O

Idea: what happens if we change a special FD to point somewhere else?

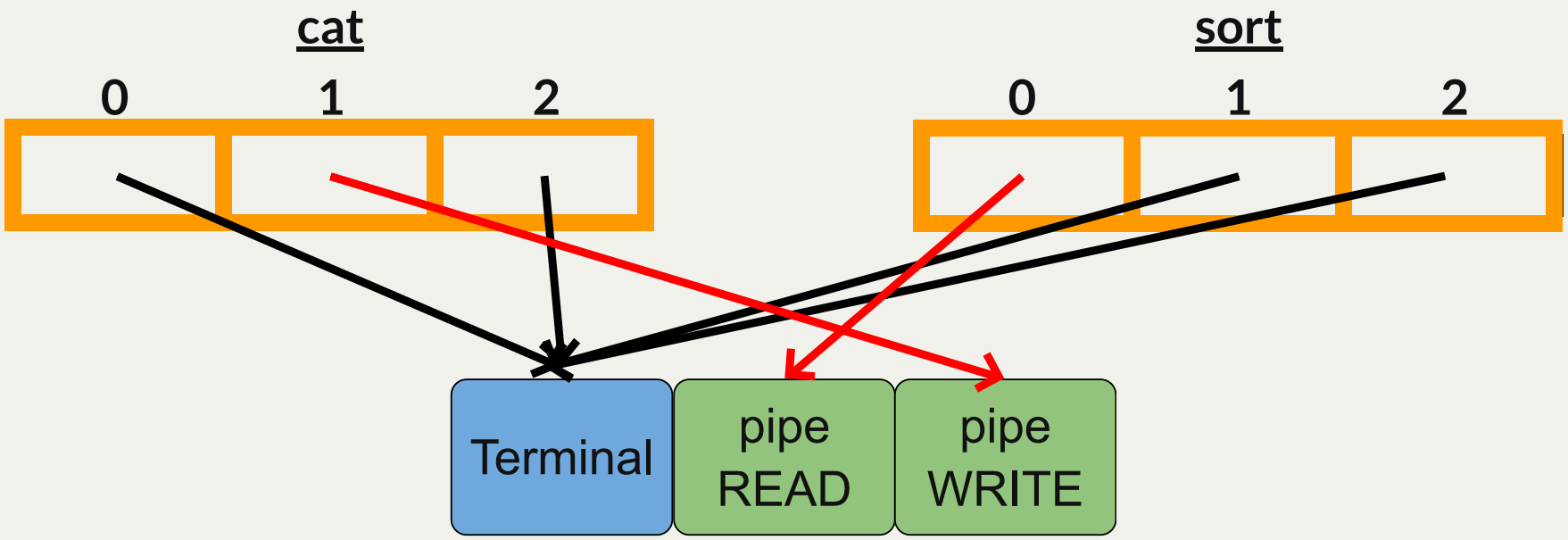
Could we do this with a pipe?



Why would this be useful?

Redirecting Process I/O

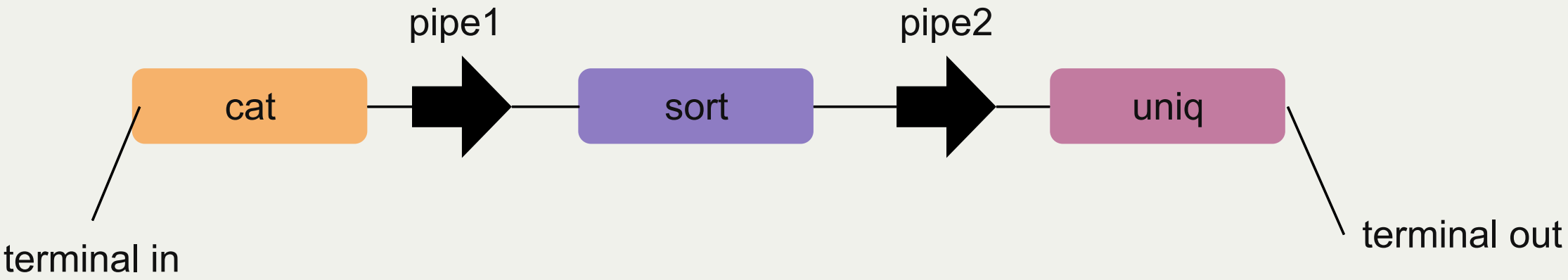
I/O redirection and pipes allow us to handle piping in our shell: e.g. **cat file.txt | sort**



Redirecting Process I/O

I/O redirection and pipes allow us to handle piping in our shell: e.g. **cat file.txt | sort | uniq**

- Shell creates three child processes: cat, uniq and sort
- Shell creates two pipes: one between cat and sort, one between sort and uniq




```
int pipe1[2];
int pipe2[2];
pipe(pipe1);
pipe(pipe2);
```

Process	stdin	stdout
cat	terminal	pipe1[1]
sort	pipe1[0]	pipe2[1]
uniq	pipe2[0]	terminal

Redirecting Process I/O

One last issue; how do we "connect" our pipe FDs to STDIN/STDOUT?

Redirecting Process I/O

One last issue; how do we "connect" our pipe FDs to STDIN/STDOUT?

dup2 makes a copy of a file descriptor entry and puts it in another file descriptor index. If the second parameter is an already-open file descriptor, it is closed before being used.

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

Redirecting Process I/O

One last issue; how do we "connect" our pipe FDs to STDIN/STDOUT?

dup2 makes a copy of a file descriptor entry and puts it in another file descriptor index. If the second parameter is an already-open file descriptor, it is closed before being used.

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

Example: we can use **dup2** to copy the pipe read file descriptor into standard input!

```
dup2(fds[0], STDIN_FILENO);
```

Redirecting Process I/O

One last issue; how do we "connect" our pipe FDs to STDIN/STDOUT?

dup2 makes a copy of a file descriptor entry and puts it in another file descriptor index. If the second parameter is an already-open file descriptor, it is closed before being used.

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

Example: we can use **dup2** to copy the pipe read file descriptor into standard input!

```
dup2(fds[0], STDIN_FILENO);
```

Second key detail: **execvp** consumes the process, ***except for the file descriptor table!***

Lecture Plan

- Review: **fork()** and **execvp()**
- Running in the background
- Introducing Pipes
- ***Practice:*** Implementing **subprocess**

subprocess File Descriptor Diagram

To practice this piping technique, let's implement a custom function called **subprocess**.

```
subprocess_t subprocess(char *command);
```

subprocess is the same as **mysystem**, except it also sets up a pipe we can use to write to the child process's STDIN.

It returns a struct containing:

- the PID of the child process
- a file descriptor we can use to write to the child's STDIN

Demo: subprocess

Lecture Recap

- Review: **fork()** and **execvp()**
- Running in the background
- Introducing Pipes
- ***Practice:*** Implementing **subprocess**

Next time: introducing signals

Practice Problems

A Publishing Error

The program below takes an arbitrary number of filenames as arguments and attempts to publish the date and time. The desired behavior is shown at right:

```c
static void publish(const char *name) {
    printf("Publishing date and time to file named \"%s\".\n", name);
    int outfile = open(name, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    dup2(outfile, STDOUT_FILENO);
    close(outfile);
    if (fork() > 0) return;
    char *argv[] = { "date", NULL };
    execvp(argv[0], argv);
}

int main(int argc, char *argv[]) {
    for (size_t i = 1; i < argc; i++) publish(argv[i]);
    return 0;
}
```

```
myth62:~$ ./publish one two three four
Publishing date and time to file named "one".
Publishing date and time to file named "two".
Publishing date and time to file named "three".
Publishing date and time to file named "four".
```

However, the program is buggy!

- What text is actually printed to standard output?
- What do each of the four files contain?
- How can we fix the issue?

publish.c

A Publishing Error

The program below takes an arbitrary number of filenames as arguments and attempts to publish the date and time. The desired behavior is shown at right:

```c
static void publish(const char *name) {
    printf("Publishing date and time to file named \"%s\".\n", name);
    int outfile = open(name, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    dup2(outfile, STDOUT_FILENO);
    close(outfile);
    if (fork() > 0) return;
    char *argv[] = { "date", NULL };
    execvp(argv[0], argv);
}

int main(int argc, char *argv[]) {
    for (size_t i = 1; i < argc; i++) publish(argv[i]);
    return 0;
}
```

```
myth62:~$ ./publish one two three four
Publishing date and time to file named "one".
Publishing date and time to file named "two".
Publishing date and time to file named "three".
Publishing date and time to file named "four".
```

However, the program is buggy!

- What text is actually printed to standard output?
- What do each of the four files contain?
- How can we fix the issue?

publish.c

A Publishing Error

The program below takes an arbitrary number of filenames as arguments and attempts to publish the date and time. The desired behavior is shown at right:

```
1  static void publish(const char *name) {
2      printf("Publishing date and time to file named \"%s\".\n", name);
3      int outfile = open(name, O_WRONLY | O_CREAT | O_TRUNC, 0644);
4      dup2(outfile, STDOUT_FILENO);
5      close(outfile);
6      if (fork() > 0) return;
7      char *argv[] = { "date", NULL };
8      execvp(argv[0], argv);
9  }
10
11 int main(int argc, char *argv[]) {
12     for (size_t i = 1; i < argc; i++) publish(argv[i]);
13     return 0;
14 }
```

```
1 myth62:~$ ./publish one two three four
2 Publishing date and time to file named "one".
3 Publishing date and time to file named "two".
4 Publishing date and time to file named "three".
4 Publishing date and time to file named "four".
5
```

However, the program is buggy!

- What text is actually printed to standard output?
- What do each of the four files contain?
- How can we fix the issue?

publish.c

A Publishing Error

The program below takes an arbitrary number of filenames as arguments and attempts to publish the date and time. The desired behavior is shown at right:

```c
static void publish(const char *name) {
    printf("Publishing date and time to file named \"%s\".\n", name);
    int outfile = open(name, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    dup2(outfile, STDOUT_FILENO);
    close(outfile);
    if (fork() > 0) return;
    char *argv[] = { "date", NULL };
    execvp(argv[0], argv);
}

int main(int argc, char *argv[]) {
    for (size_t i = 1; i < argc; i++) publish(argv[i]);
    return 0;
}
```

```
myth62:~$ ./publish one two three four
Publishing date and time to file named "one".
Publishing date and time to file named "two".
Publishing date and time to file named "three".
Publishing date and time to file named "four".
```

However, the program is buggy!

- What text is actually printed to standard output?
- What do each of the four files contain?
- How can we fix the issue?

publish.c

A Publishing Error

The program below takes an arbitrary number of filenames as arguments and attempts to publish the date and time. The desired behavior is shown at right:

```
1  static void publish(const char *name) {
2      printf("Publishing date and time to file named \"%s\".\n", name);
3      int outfile = open(name, O_WRONLY | O_CREAT | O_TRUNC, 0644);
4      dup2(outfile, STDOUT_FILENO);
5      close(outfile);
6      if (fork() > 0) return;
7      char *argv[] = { "date", NULL };
8      execvp(argv[0], argv);
9  }
10
11 int main(int argc, char *argv[]) {
12     for (size_t i = 1; i < argc; i++) publish(argv[i]);
13     return 0;
14 }
```

```
1 myth62:~$ ./publish one two three four
2 Publishing date and time to file named "one".
3 Publishing date and time to file named "two".
4 Publishing date and time to file named "three".
4 Publishing date and time to file named "four".
5
```

However, the program is buggy!

- What text is actually printed to standard output?
- What do each of the four files contain?
- How can we fix the issue?

publish.c

A Publishing Error

The program below takes an arbitrary number of filenames as arguments and attempts to publish the date and time. The desired behavior is shown at right:

```c
static void publish(const char *name) {
    printf("Publishing date and time to file named \"%s\".\n", name);
    int outfile = open(name, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    dup2(outfile, STDOUT_FILENO);
    close(outfile);
    if (fork() > 0) return;
    char *argv[] = { "date", NULL };
    execvp(argv[0], argv);
}

int main(int argc, char *argv[]) {
    for (size_t i = 1; i < argc; i++) publish(argv[i]);
    return 0;
}
```

```
myth62:~$ ./publish one two three four
Publishing date and time to file named "one".
Publishing date and time to file named "two".
Publishing date and time to file named "three".
Publishing date and time to file named "four".
```

However, the program is buggy!

- What text is actually printed to standard output?
- What do each of the four files contain?
- How can we fix the issue?

publish.c

A Publishing Error

Because the child processes (and only the child processes) should be redirecting, we should open, dup2, and close in child-specific code. A happy side effect of the change is that we never muck with STDOUT_FILENO in the parent if we confine the redirection code to the child.

Solution:

```
1 static void publish(const char *name) {
2     printf("Publishing date and time to file named \"%s\".\n", name);
3     if (fork() > 0) return;
4     int outfile = open(name, O_WRONLY | O_CREAT | O_TRUNC, 0644);
5     dup2(outfile, STDOUT_FILENO);
6     close(outfile);
7     char *argv[] = { "date", NULL };
8     execvp(argv[0], argv);
9 }
```

publish.c

A Publishing Error

Because the child processes (and only the child processes) should be redirecting, we should open, dup2, and close in child-specific code. A happy side effect of the change is that we never muck with STDOUT_FILENO in the parent if we confine the redirection code to the child.

Solution:

```
1 static void publish(const char *name) {
2     printf("Publishing date and time to file named \"%s\".\n", name);
3     if (fork() > 0) return;
4     int outfile = open(name, O_WRONLY | O_CREAT | O_TRUNC, 0644);
5     dup2(outfile, STDOUT_FILENO);
6     close(outfile);
7     char *argv[] = { "date", NULL };
8     execvp(argv[0], argv);
9 }
```

publish.c

A Publishing Error

Because the child processes (and only the child processes) should be redirecting, we should open, dup2, and close in child-specific code. A happy side effect of the change is that we never muck with STDOUT_FILENO in the parent if we confine the redirection code to the child.

Solution:

```
1 static void publish(const char *name) {
2     printf("Publishing date and time to file named \"%s\".\n", name);
3     if (fork() > 0) return;
4     int outfile = open(name, O_WRONLY | O_CREAT | O_TRUNC, 0644);
5     dup2(outfile, STDOUT_FILENO);
6     close(outfile);
7     char *argv[] = { "date", NULL };
8     execvp(argv[0], argv);
9 }
```

publish.c

A Publishing Error

Because the child processes (and only the child processes) should be redirecting, we should open, dup2, and close in child-specific code. A happy side effect of the change is that we never muck with STDOUT_FILENO in the parent if we confine the redirection code to the child.

Solution:

```
1 static void publish(const char *name) {
2     printf("Publishing date and time to file named \"%s\".\n", name);
3     if (fork() > 0) return;
4     int outfile = open(name, O_WRONLY | O_CREAT | O_TRUNC, 0644);
5     dup2(outfile, STDOUT_FILENO);
6     close(outfile);
7     char *argv[] = { "date", NULL };
8     execvp(argv[0], argv);
9 }
```

publish.c

A Publishing Error

Because the child processes (and only the child processes) should be redirecting, we should open, dup2, and close in child-specific code. A happy side effect of the change is that we never muck with STDOUT_FILENO in the parent if we confine the redirection code to the child.

Solution:

```
1 static void publish(const char *name) {
2     printf("Publishing date and time to file named \"%s\".\n", name);
3     if (fork() > 0) return;
4     int outfile = open(name, O_WRONLY | O_CREAT | O_TRUNC, 0644);
5     dup2(outfile, STDOUT_FILENO);
6     close(outfile);
7     char *argv[] = { "date", NULL };
8     execvp(argv[0], argv);
9 }
```

publish.c

A Publishing Error

Because the child processes (and only the child processes) should be redirecting, we should open, dup2, and close in child-specific code. A happy side effect of the change is that we never muck with STDOUT_FILENO in the parent if we confine the redirection code to the child.

Solution:

```
1 static void publish(const char *name) {
2     printf("Publishing date and time to file named \"%s\".\n", name);
3     if (fork() > 0) return;
4     int outfile = open(name, O_WRONLY | O_CREAT | O_TRUNC, 0644);
5     dup2(outfile, STDOUT_FILENO);
6     close(outfile);
7     char *argv[] = { "date", NULL };
8     execvp(argv[0], argv);
9 }
```

publish.c

captureProcess

Let's implement a custom function called **captureProcess**, like **subprocess** except instead of setting up a pipe to write to the child's STDIN, it's a pipe to read from its STDOUT.

```
subprocess_t captureProcess(char *command);
```

It returns a struct containing:

- the PID of the child process
- a file descriptor we can use to **read from the child's STDOUT**

captureProcess

Let's implement a custom function called **captureProcess**, like **subprocess** except instead of setting up a pipe to write to the child's STDIN, it's a pipe to read from its

```
1  subprocess_t captureProcess(char *command) {
2      int fds[2];
3      pipe(fds);
4
5      pid_t pidOrZero = fork();
6      if (pidOrZero == 0) {
7          // We are not reading from the pipe, only writing to it
8          close(fds[0]);
9
10         // Duplicate the write end of the pipe into STDOUT
11         dup2(fds[1], STDOUT_FILENO);
12         close(fds[1]);
13
14         char *arguments[] = {"/bin/sh", "-c", command, NULL};
15         execvp(arguments[0], arguments);
16         exitIf(true, kExecFailed, stderr, "execvp failed to invoke this: %s.\n", command);
17     }
18
19     close(fds[1]);
20     return (subprocess_t) { pidOrZero, fds[0] };
21 }
```

captureProcess.c