# CS110 Lecture 9: Pipes and Interprocess Communication, Part 2

**CS110: Principles of Computer Systems**

Winter 2021-2022

Stanford University

**Instructors**: Nick Troccoli and Jerry Cain

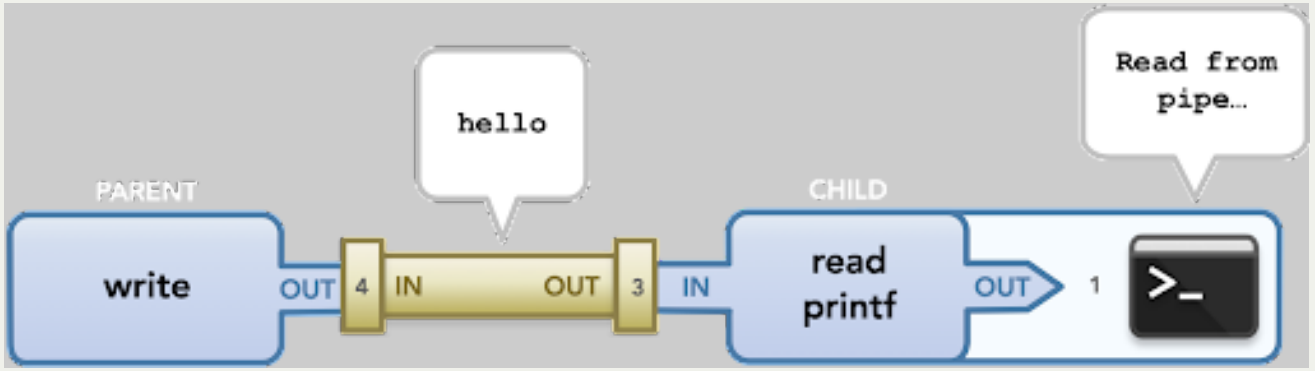
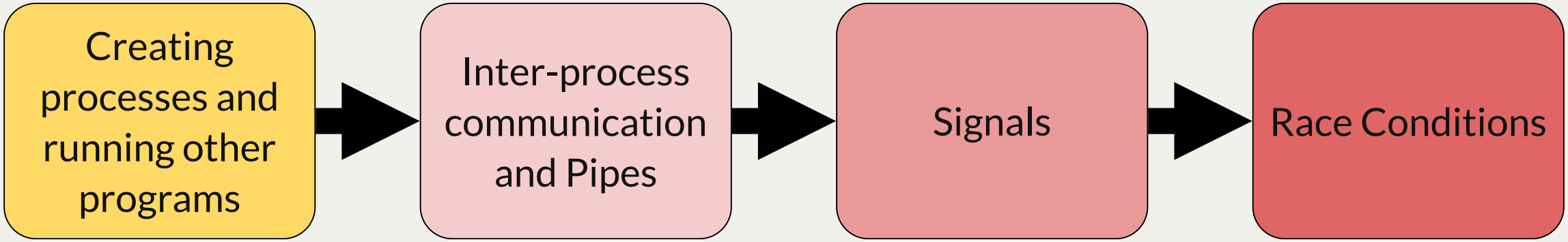



*Illustration courtesy of Roz Cyrus.*

PDF of this presentation

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

# Learning About Processes




**Lecture 6/7** | **This lecture** | **Lecture 10/11** | **Lecture 11**

assign3: implement multiprocessing programs like "trace" (to trace another program's behavior) and "farm" (parallelize tasks)

assign4: implement your own shell!

# Learning Goals

- Get more practice creating and using pipes
- Learn about **dup2** to create and manipulate file descriptors
- Use pipes to redirect process input and output

# Lecture Plan

- Review: pipes
- Redirecting process I/O
- ***Practice:*** Implementing **subprocess**
- ***Practice:*** Implementing **pipeline**

# Lecture Plan

- **Review: pipes**
- Redirecting process I/O
- ***Practice:*** Implementing **subprocess**
- ***Practice:*** Implementing **pipeline**

# Pipes

- A pipe is a set of two file descriptors representing a "virtual file" that can be written to and read from
- It's *not actually a physical file on disk* - we are just using files as an abstraction
- Any data you write to the write FD can be read from the read FD
- Because file descriptors are duplicated on **fork()**, we can create pipes that are shared across processes!

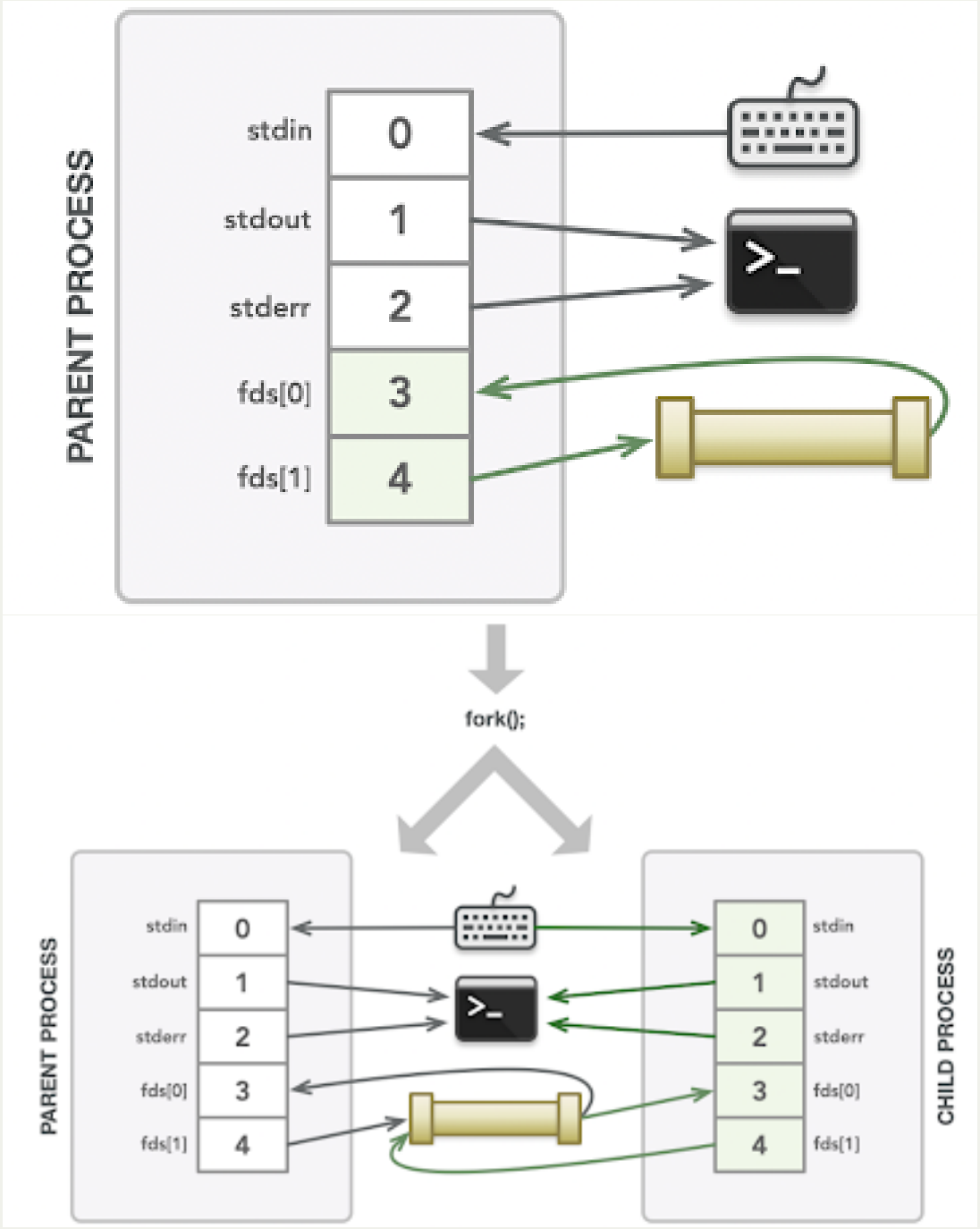



*Illustration courtesy of Roz Cyrus.*

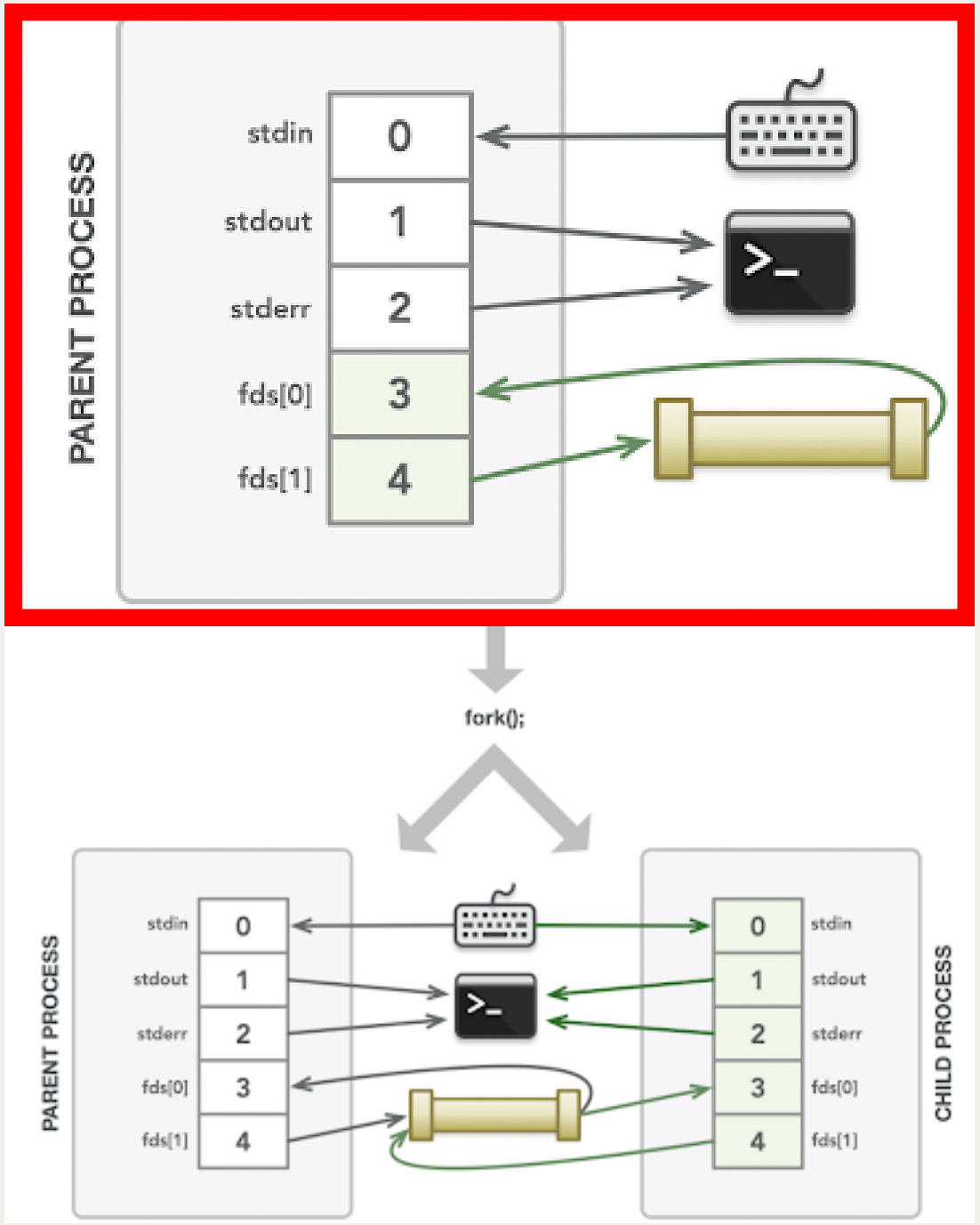
**Key Idea:** because the pipe file descriptors are duplicated in the child, we need to close the 2 pipe ends ***in both the parent and the child.***

# Parent-Child Communication

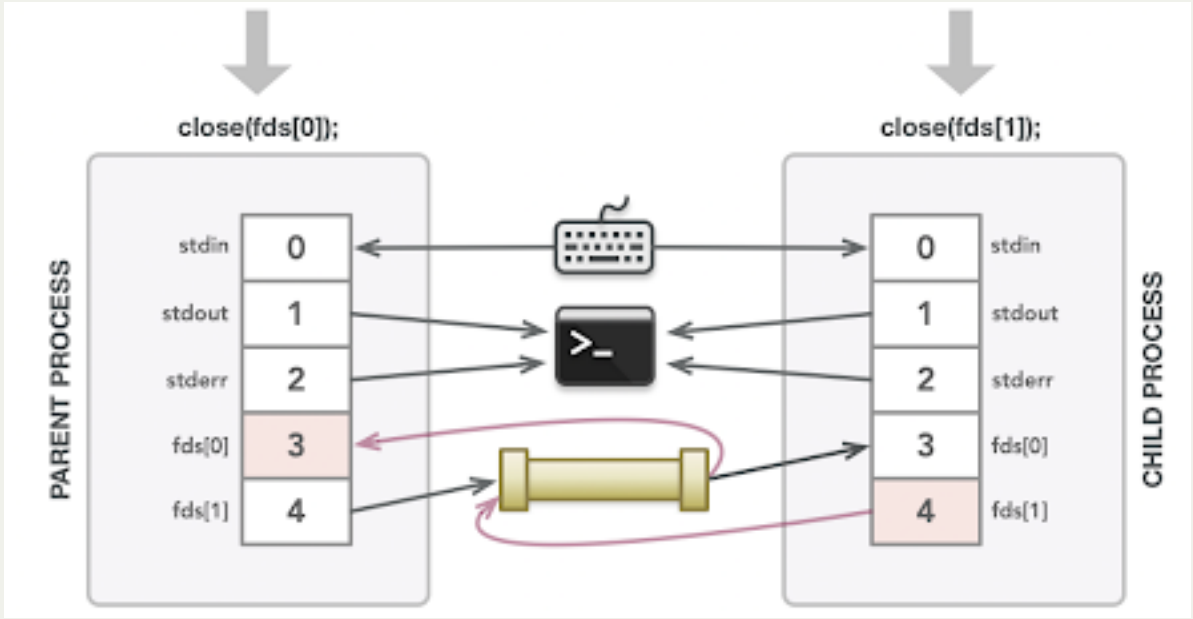
```c
static const char * kPipeMessage = "Hello, this message is coming through a pipe.";
int main(int argc, char *argv[]) {
    int fds[2];
    pipe(fds);
    size_t bytesSent = strlen(kPipeMessage) + 1;

    pid_t pidOrZero = fork();
    if (pidOrZero == 0) {
        // In the child, we only read from the pipe
        close(fds[1]);
        char buffer[bytesSent];
        read(fds[0], buffer, sizeof(buffer));
        close(fds[0]);
        printf("Message from parent: %s\n", buffer);
        return 0;
    }

    // In the parent, we only write to the pipe (assume everything is written)
    close(fds[0]);
    write(fds[1], kPipeMessage, bytesSent);
    close(fds[1]);
    waitpid(pidOrZero, NULL, 0);
    return 0;
}
```

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

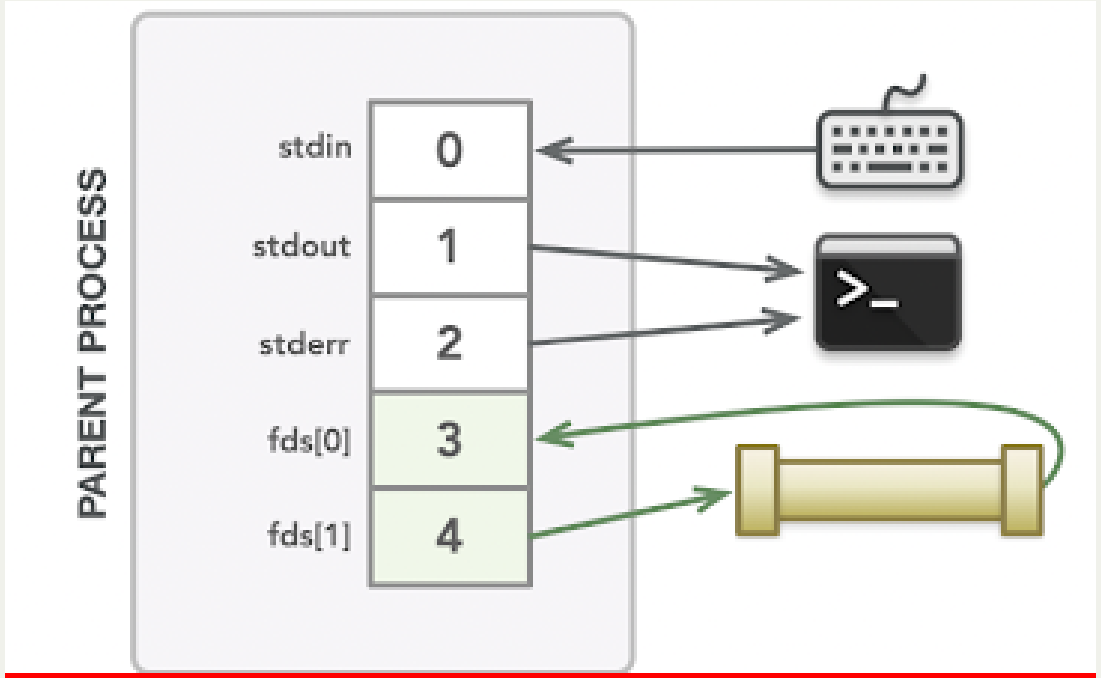
parent-child-pipe.c

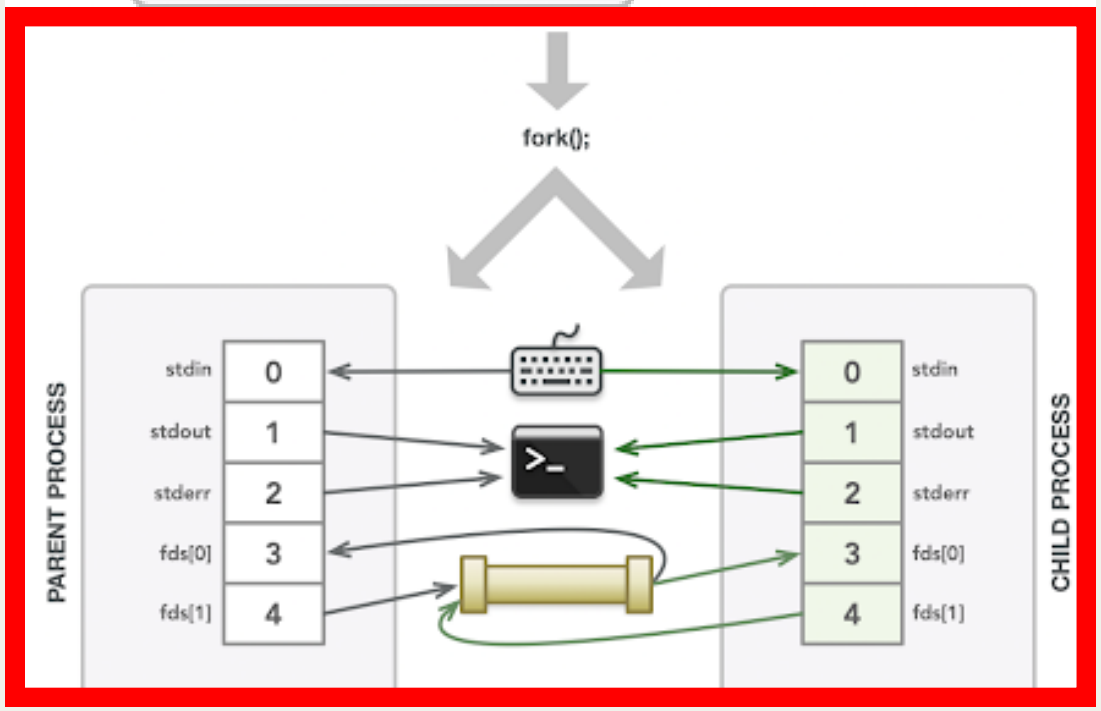



*continued...*

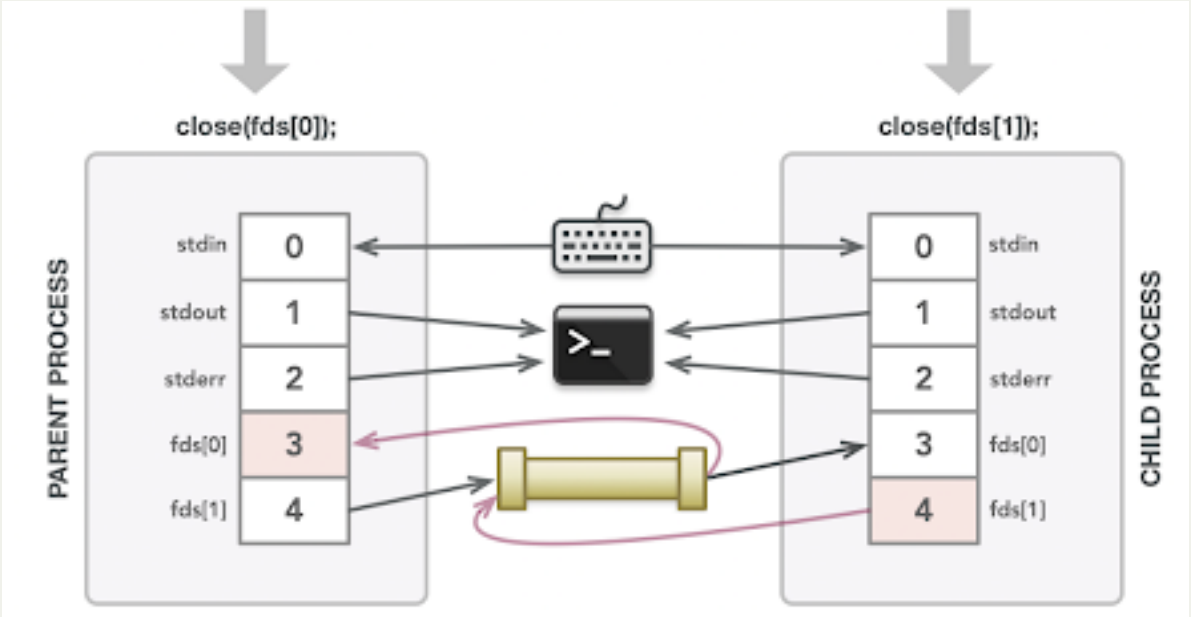



*Illustrations courtesy of Roz Cyrus.*

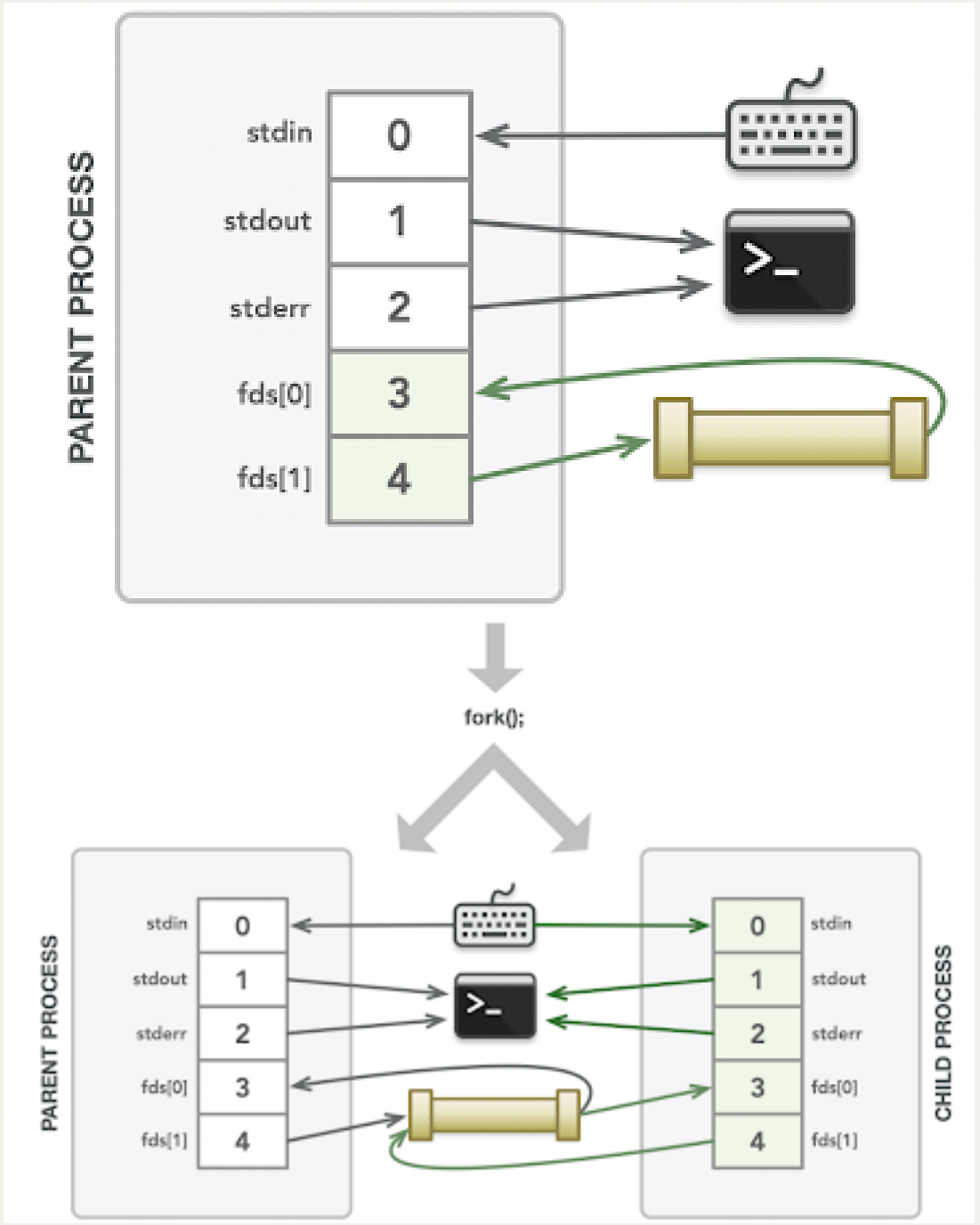


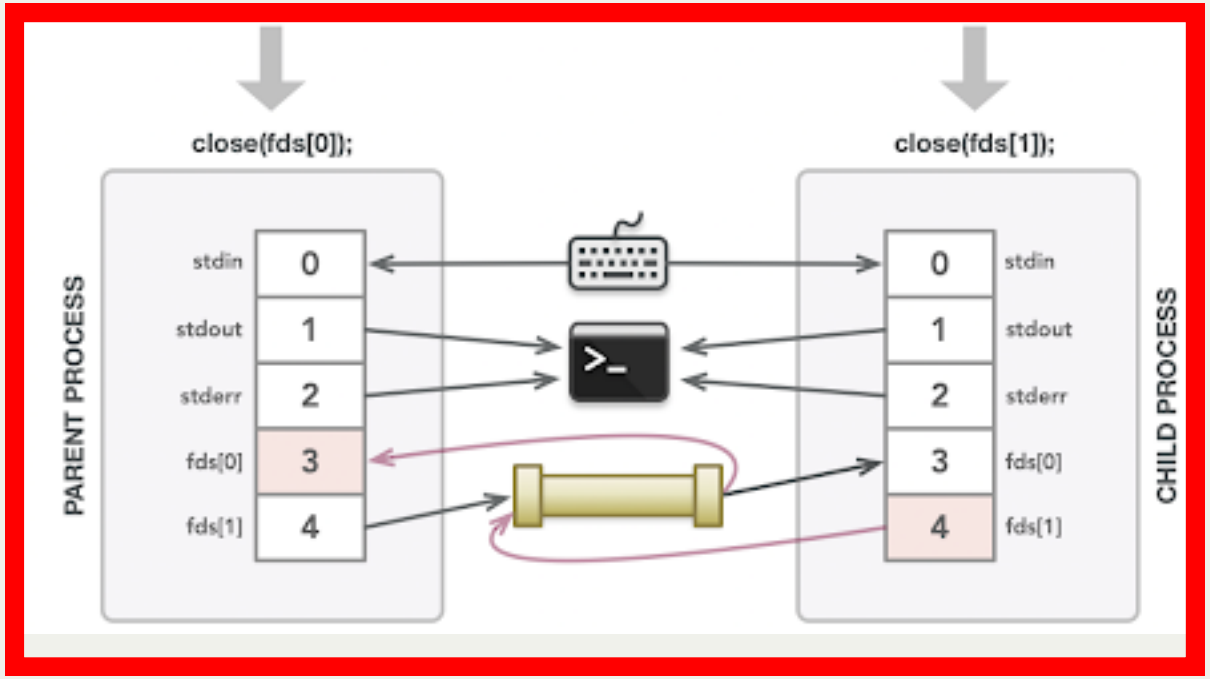

*continued...*



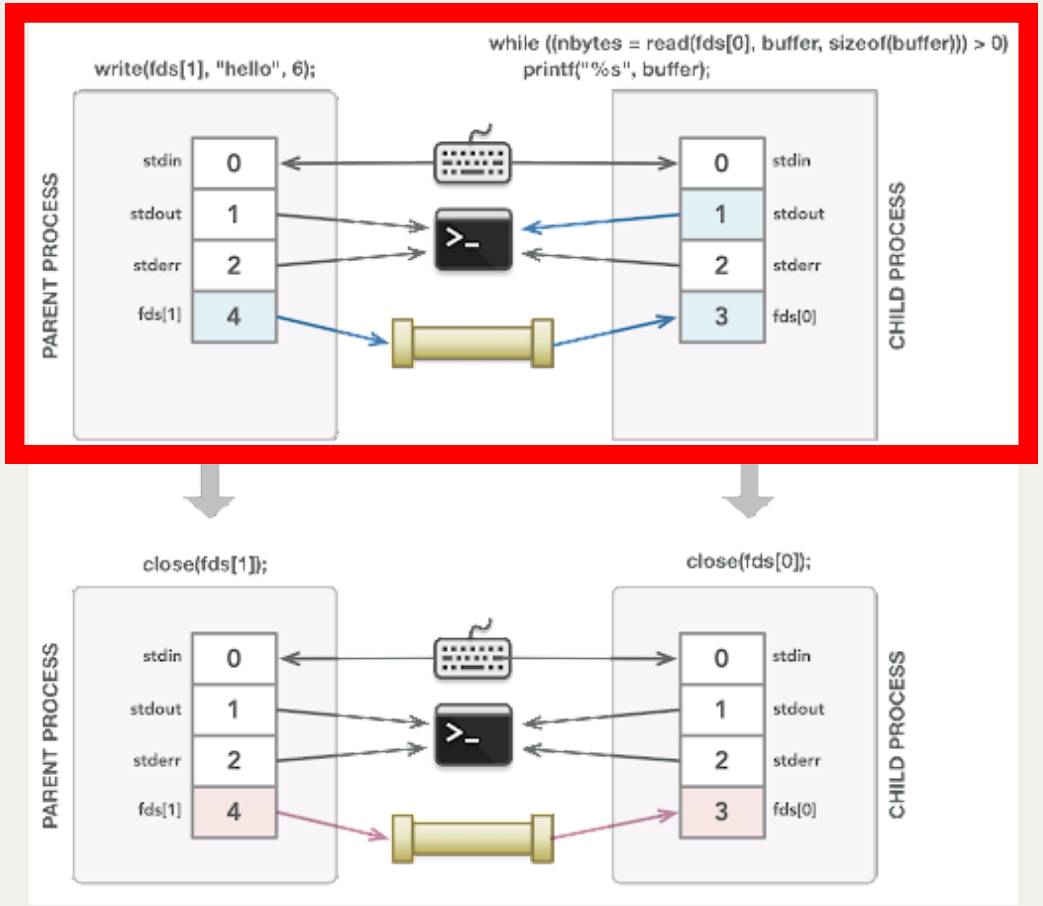

*Illustrations courtesy of Roz Cyrus.*



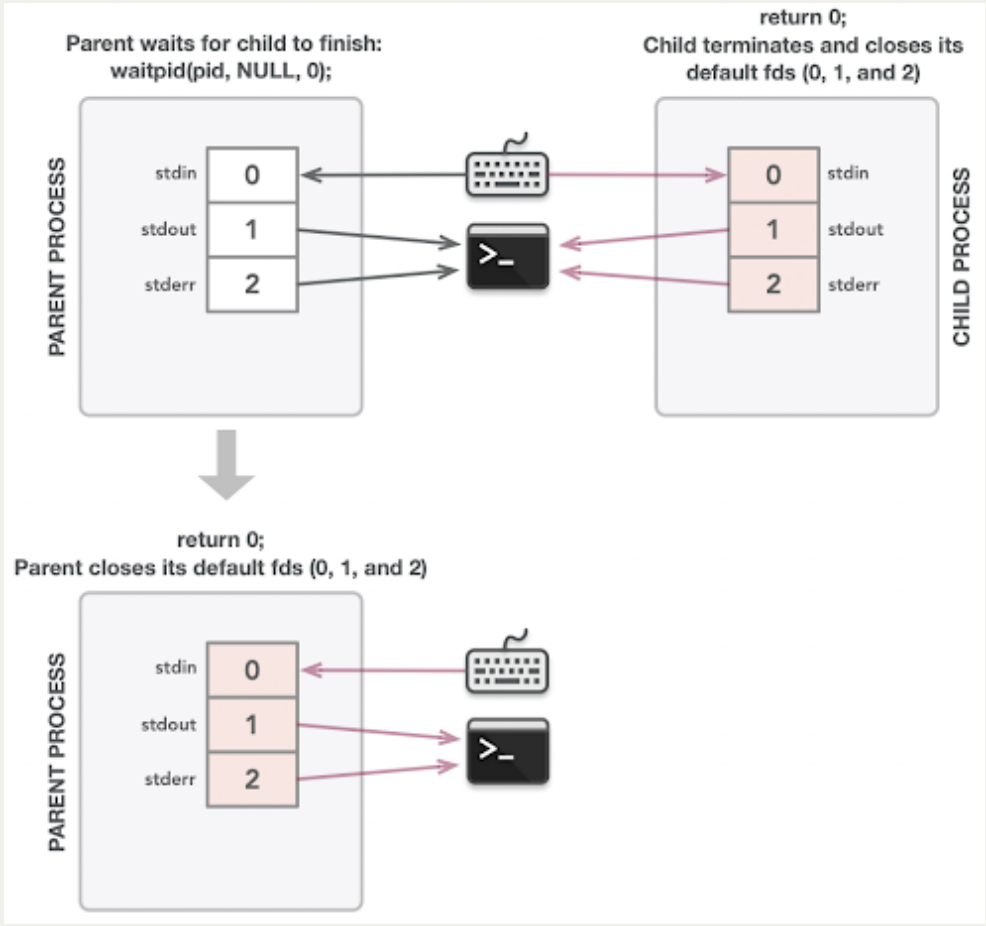

*continued...*



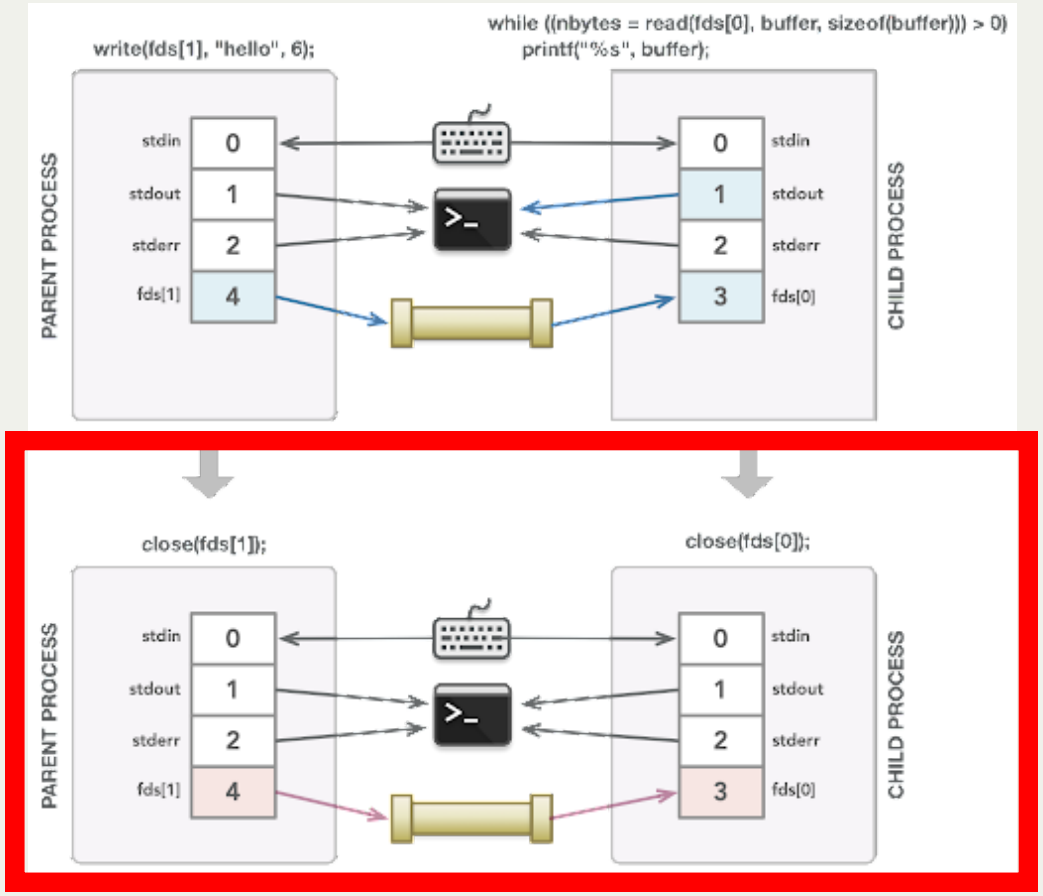

*Illustrations courtesy of Roz Cyrus.*



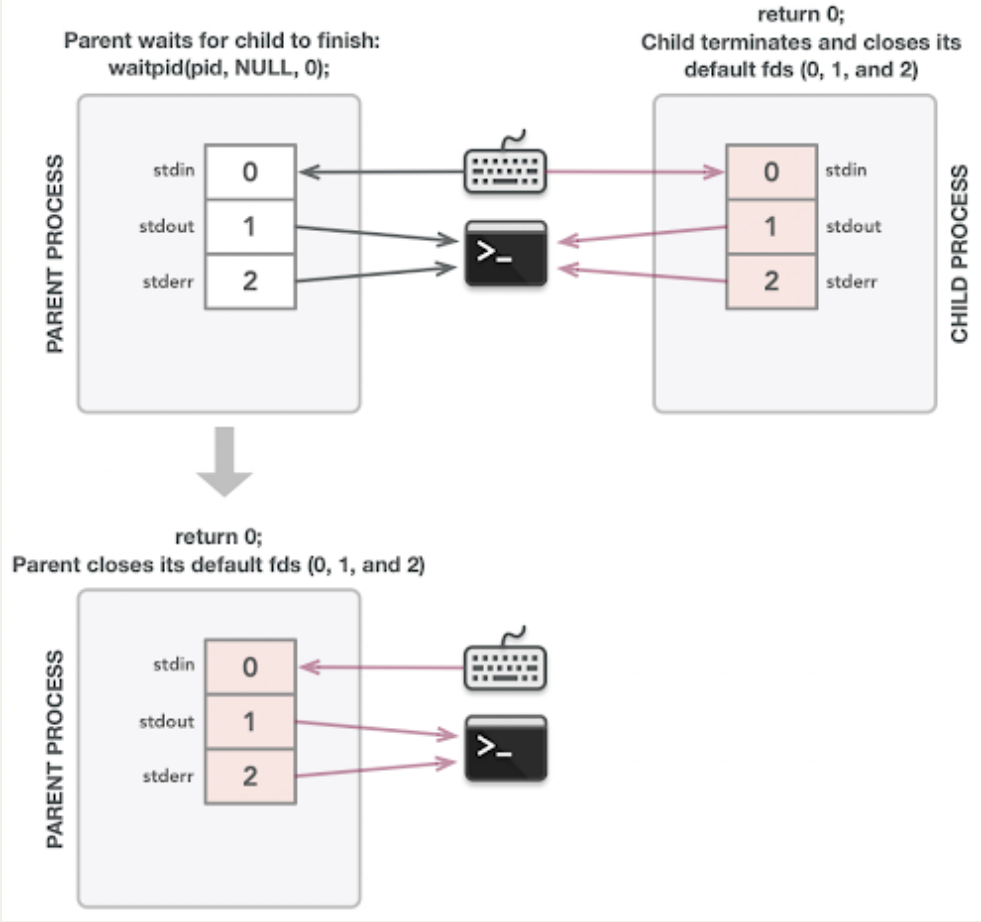

*continued...*



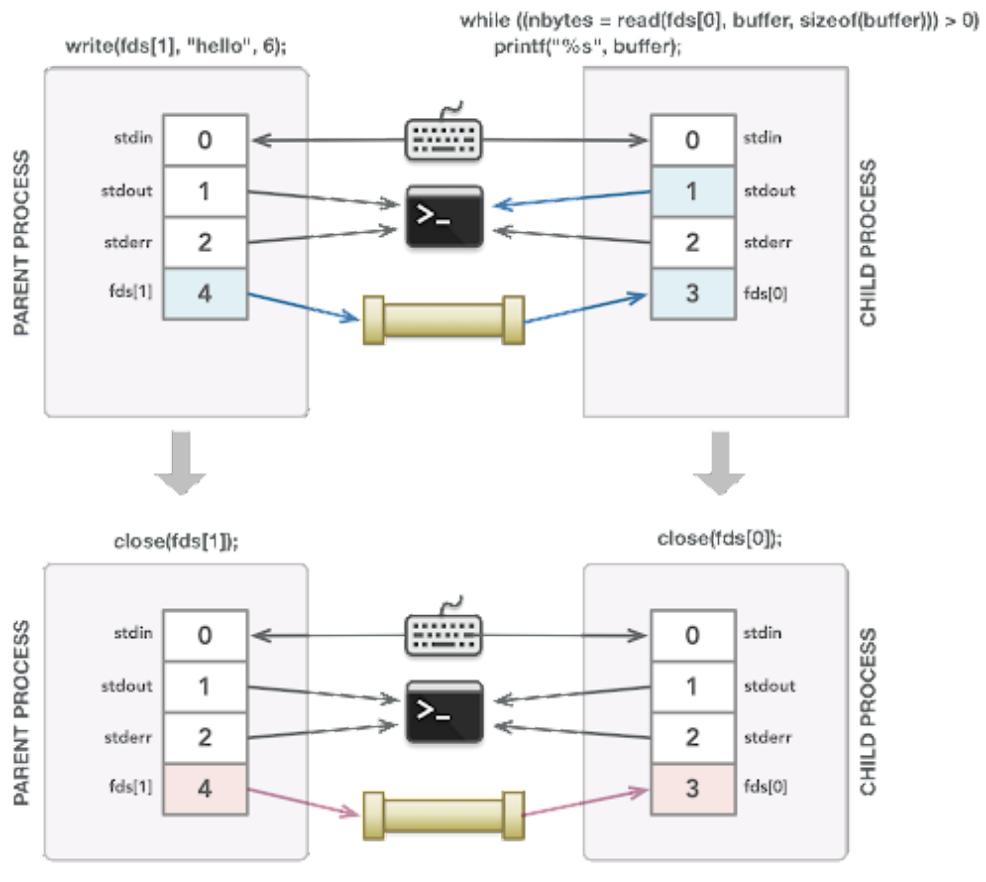

*Illustrations courtesy of Roz Cyrus.*



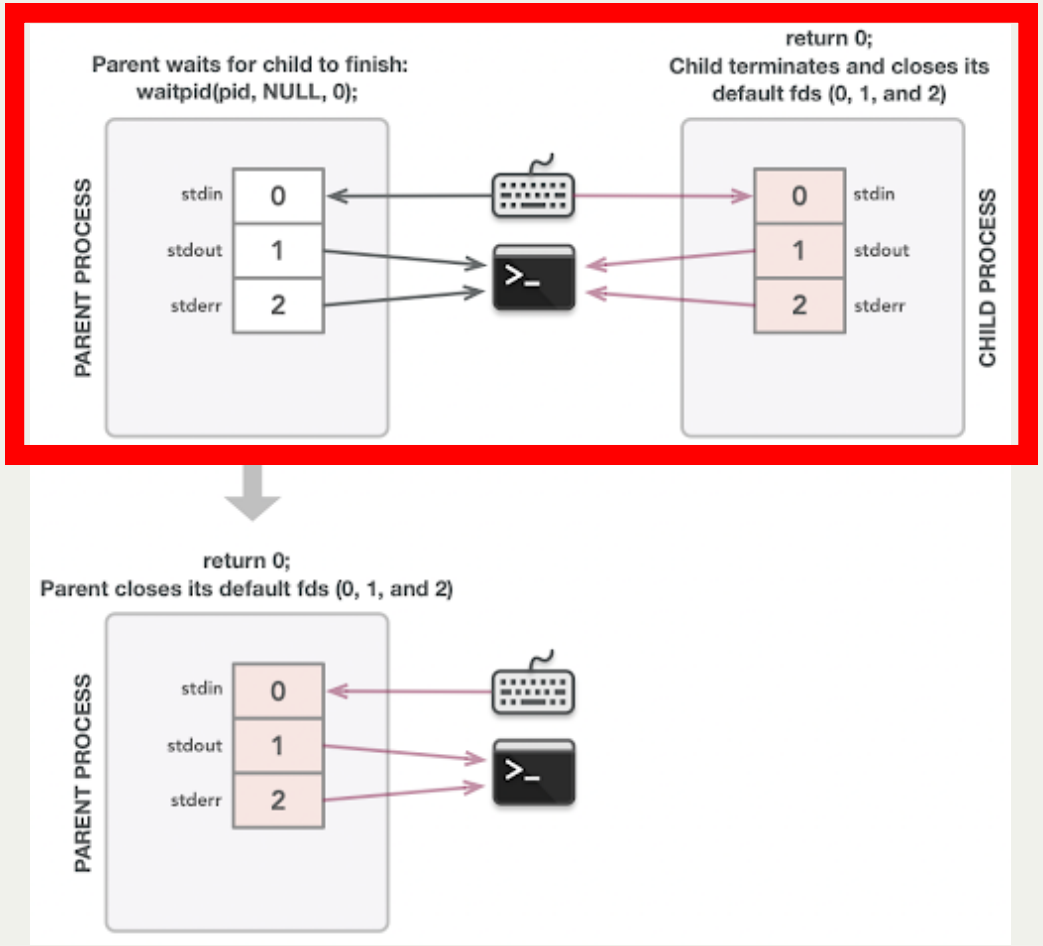

*continued...*



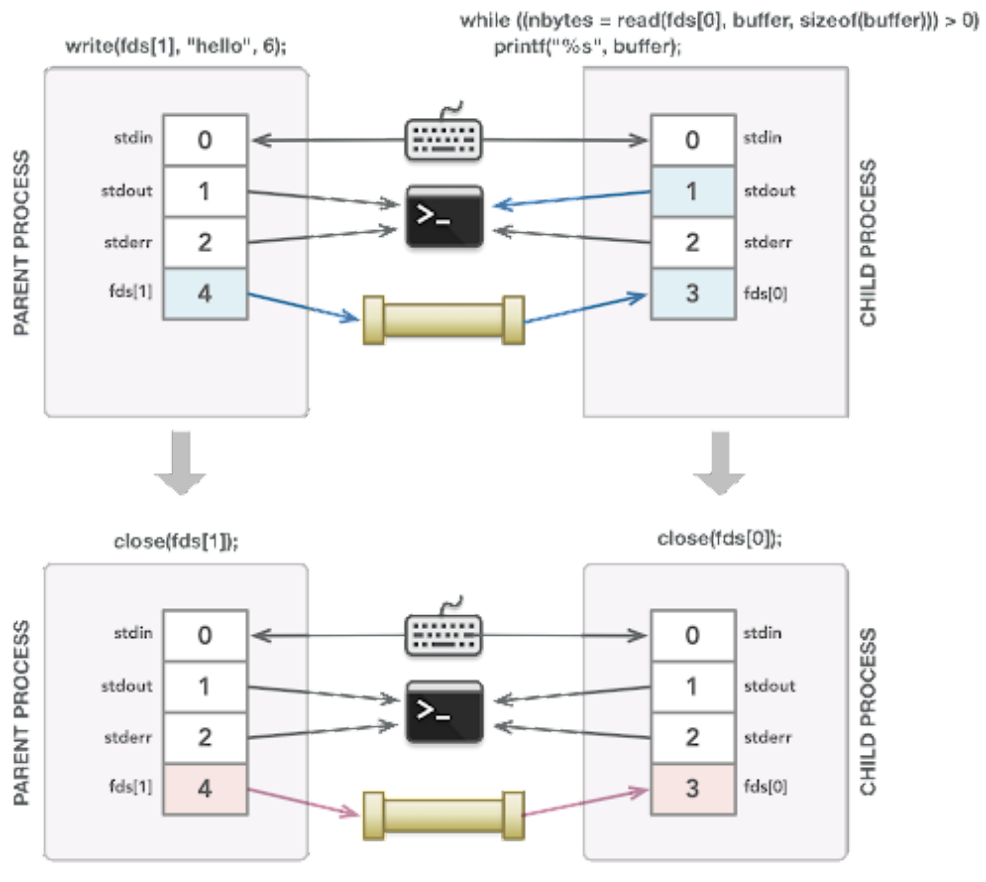

*Illustrations courtesy of Roz Cyrus.*



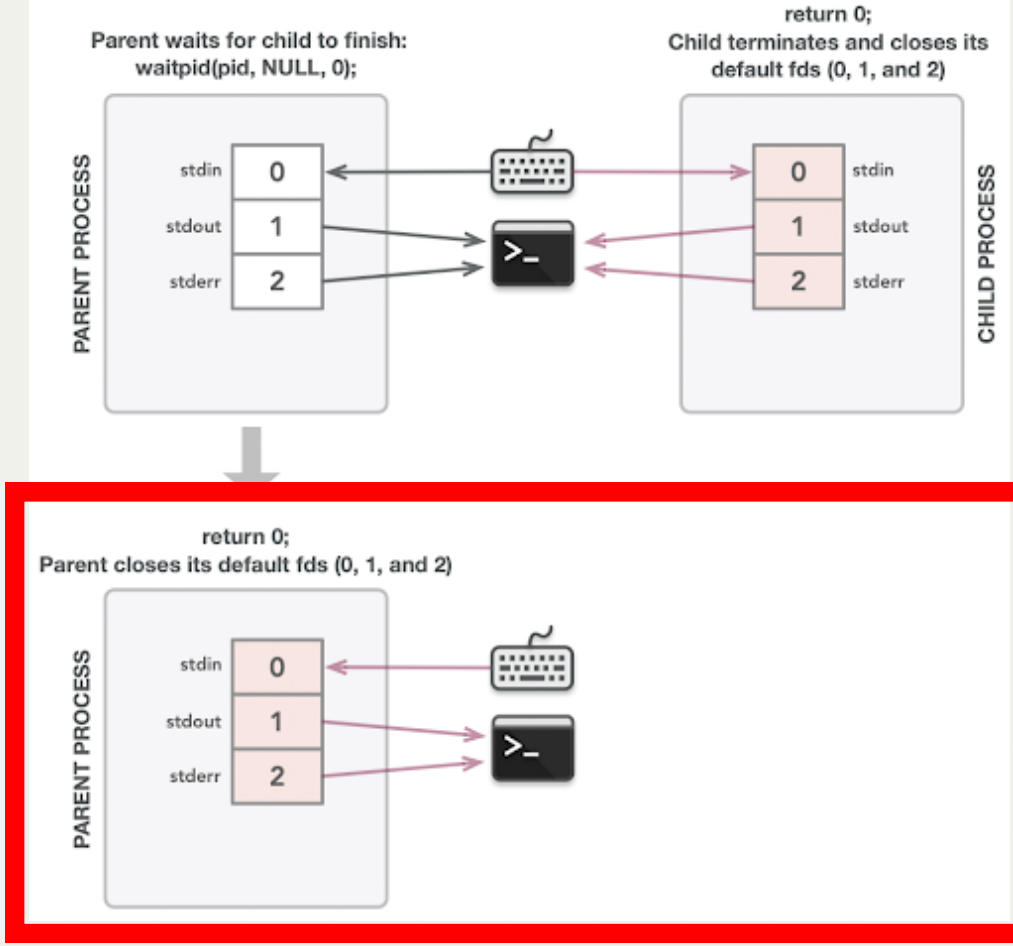

*continued...*




*Illustrations courtesy of Roz Cyrus.*




*continued...*




*Illustrations courtesy of Roz Cyrus.*

# 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

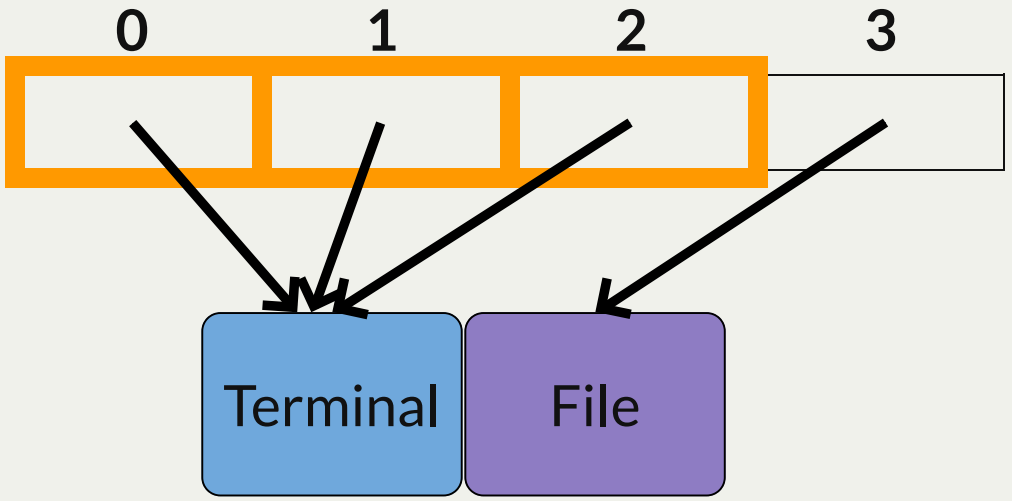
This is the core idea behind how a shell can support piping between processes
(e.g. **cat file.txt | uniq | sort**).

# Lecture Plan

- Review: pipes
- **Redirecting process I/O**
- ***Practice:*** Implementing **subprocess**
- ***Practice:*** Implementing **pipeline**

# 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).

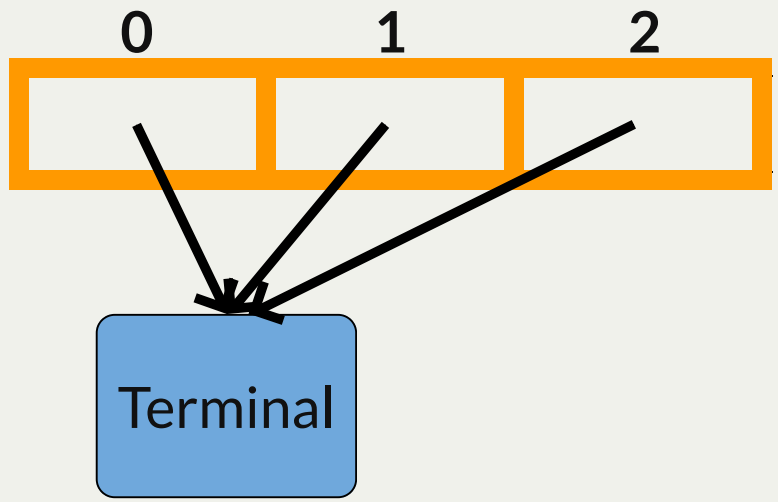



**Idea:** what happens if we change FD 1 to point somewhere else?

# Redirecting Process I/O

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

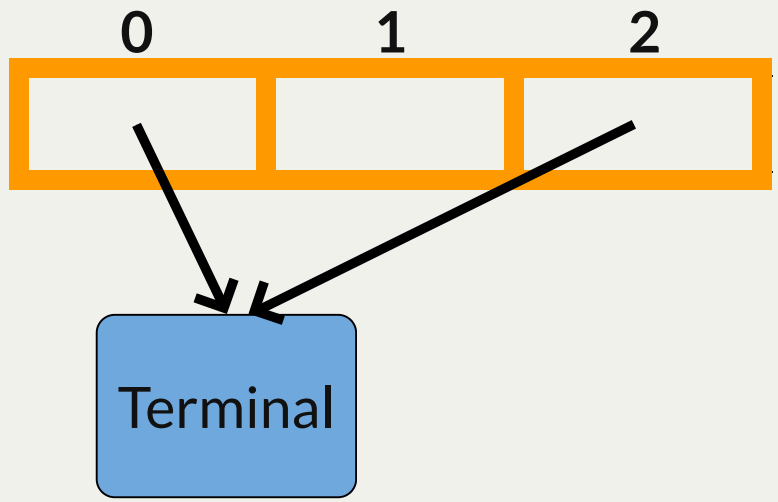
```c
int main() {
    printf("This will print to the terminal\n");
    close(STDOUT_FILENO);

    // fd will always be 1
    int fd = open("myfile.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);

    printf("This will print to myfile.txt!\n");
    close(fd);
    return 0;
}
```



# Redirecting Process I/O

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

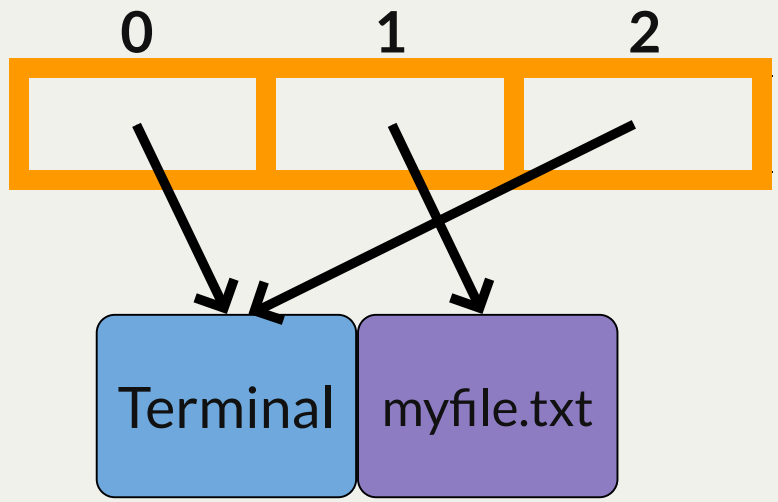
```c
int main() {
    printf("This will print to the terminal\n");
    close(STDOUT_FILENO);

    // fd will always be 1
    int fd = open("myfile.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);

    printf("This will print to myfile.txt!\n");
    close(fd);
    return 0;
}
```



# Redirecting Process I/O

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

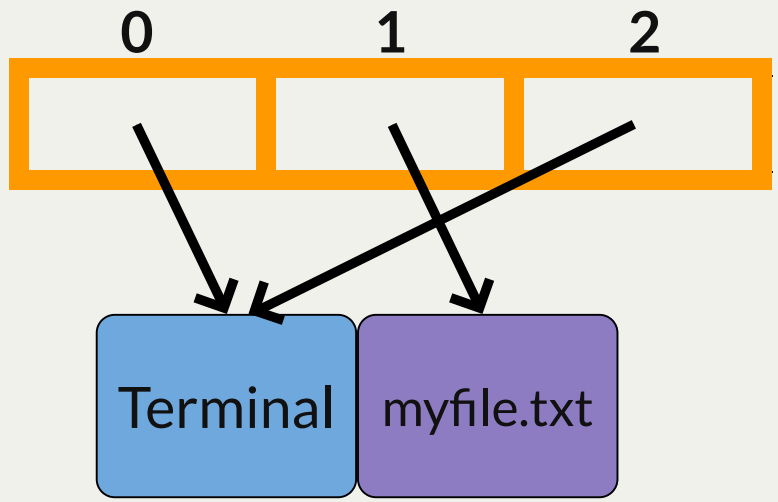
```c
int main() {
    printf("This will print to the terminal\n");
    close(STDOUT_FILENO);

    // fd will always be 1
    int fd = open("myfile.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);

    printf("This will print to myfile.txt!\n");
    close(fd);
    return 0;
}
```



# Redirecting Process I/O

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

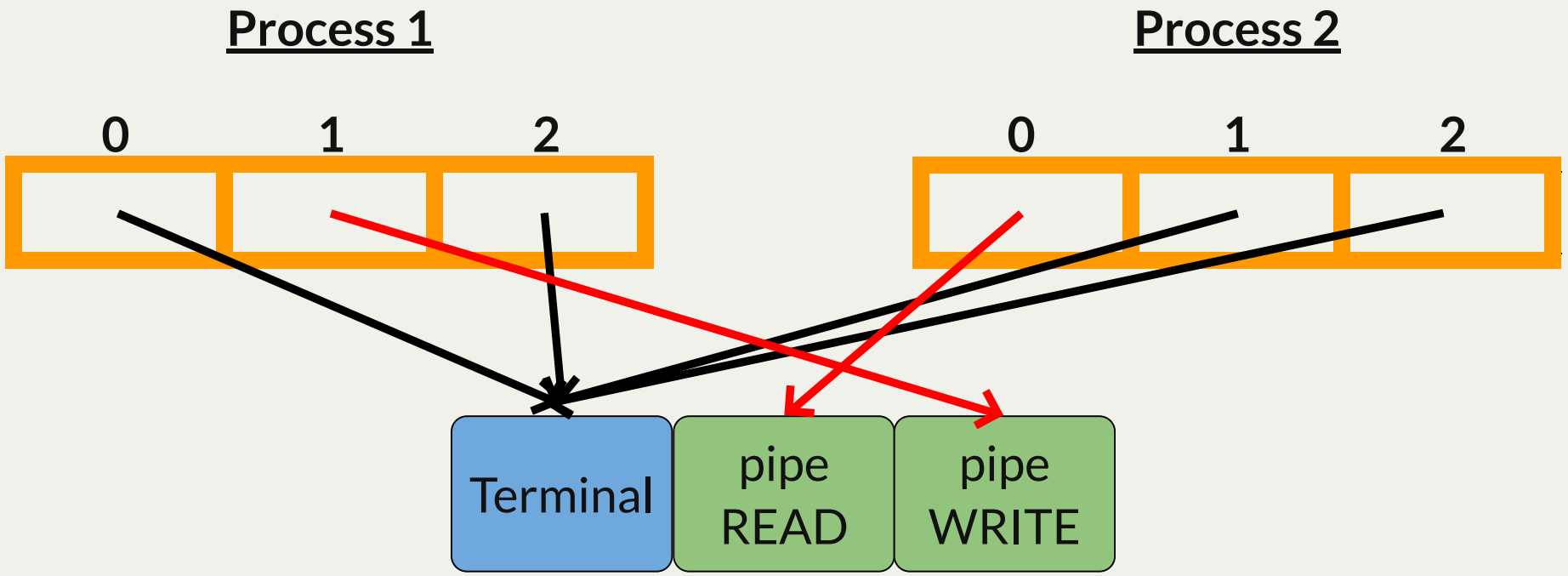
```c
int main() {
    printf("This will print to the terminal\n");
    close(STDOUT_FILENO);

    // fd will always be 1
    int fd = open("myfile.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);

    printf("This will print to myfile.txt!\n");
    close(fd);
    return 0;
}
```



# Redirecting Process I/O

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

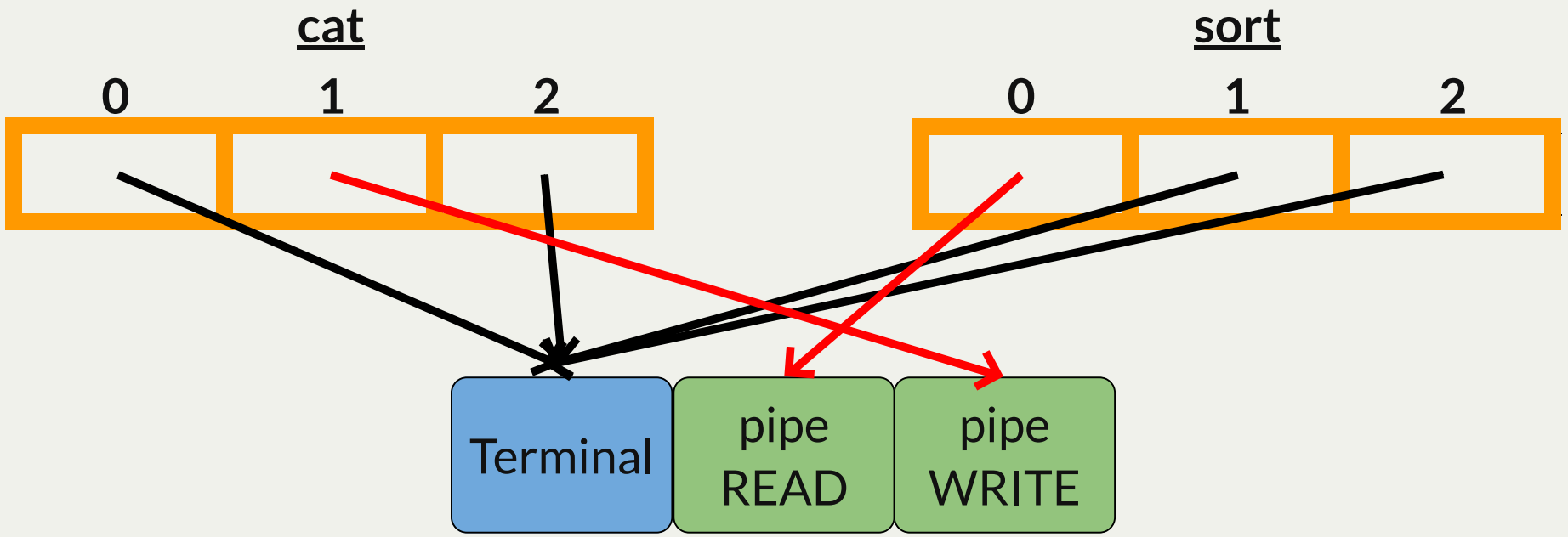
**Could we do this with a pipe?**




**Why would this be useful?**

# Redirecting Process I/O

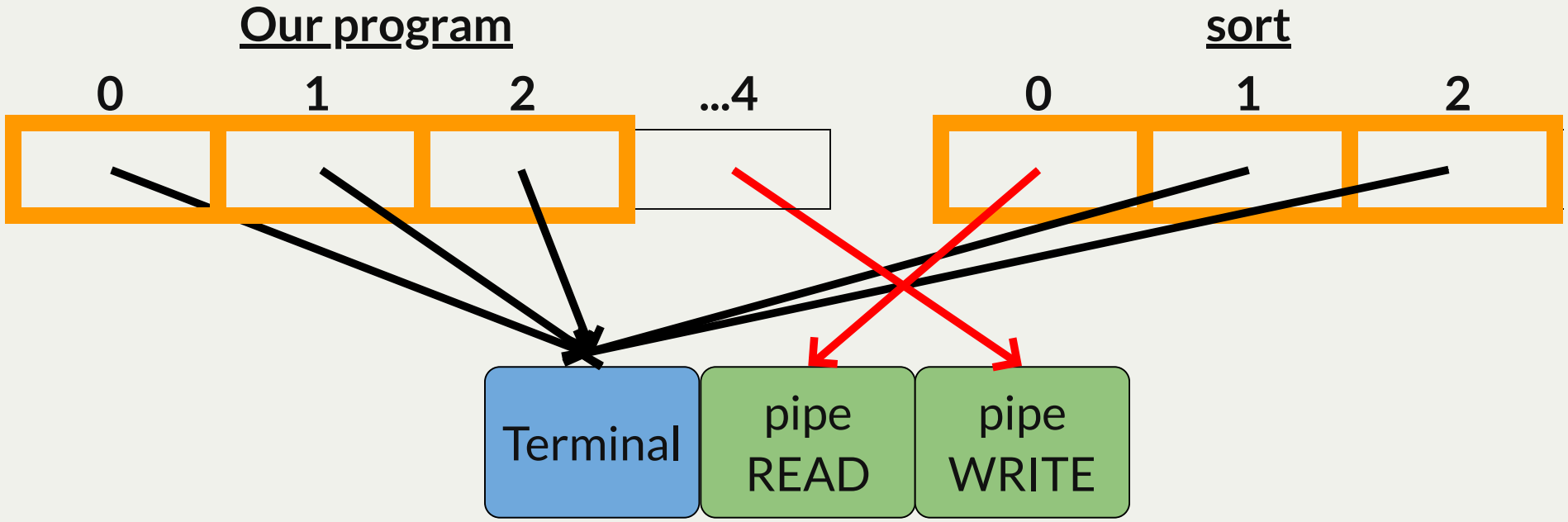
I/O redirection and pipes allow us to handle piping in our shell: e.g. **cat file.txt | sort**




This allows the shell to link together two distinct executables without them knowing. (How?)

# Redirecting Process I/O

**Stepping stone:** our first goal is to write a program that spawns another program and sends data to its STDIN.




The **sort** executable has no idea its input is not coming from terminal entry!

# Redirecting Process I/O

Our first goal is to write a program that spawns another program and sends data to its STDIN.

1. Our program creates a pipe
2. Our program spawns a child process
3. That child process changes its STDIN to be the pipe read end (how?)
4. That child process calls **execvp** to run the specified command
5. The parent writes to the write end of the pipe, which appears to the child as its STDIN

"Wait a minute...I thought execvp consumed the process? How do the file descriptors stick around?"
**New insight: execvp** consumes the process, but *leaves the file descriptor table in tact!*

# Redirecting Process I/O

One 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);
```

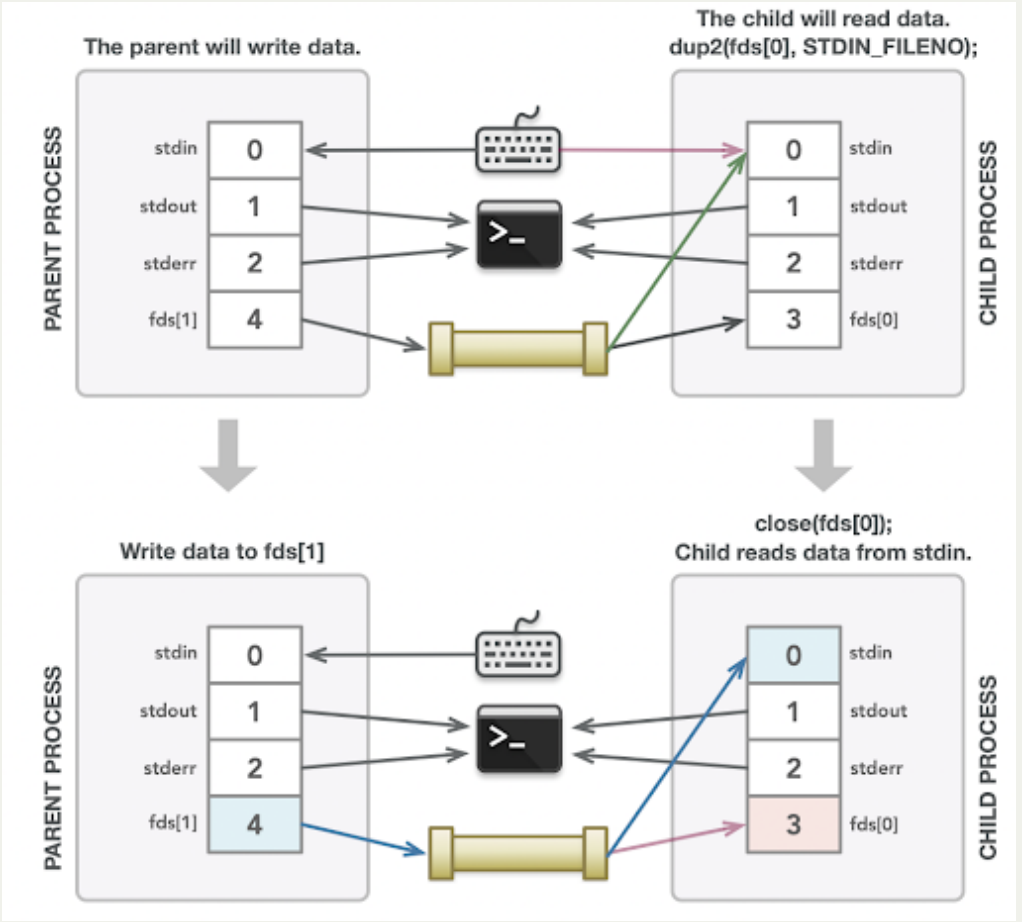
Example: we can use **dup2** to copy the pipe read file descriptor into standard input!

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

# Redirecting Process I/O

**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);
```




*Illustrations courtesy of Roz Cyrus.*

# Lecture Plan

- Review: our first shell
- Running in the background
- Introducing Pipes
  - What are pipes?
  - Pipes between processes
  - Redirecting process I/O
- ***Practice:*** **Implementing subprocess**

# subprocess

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

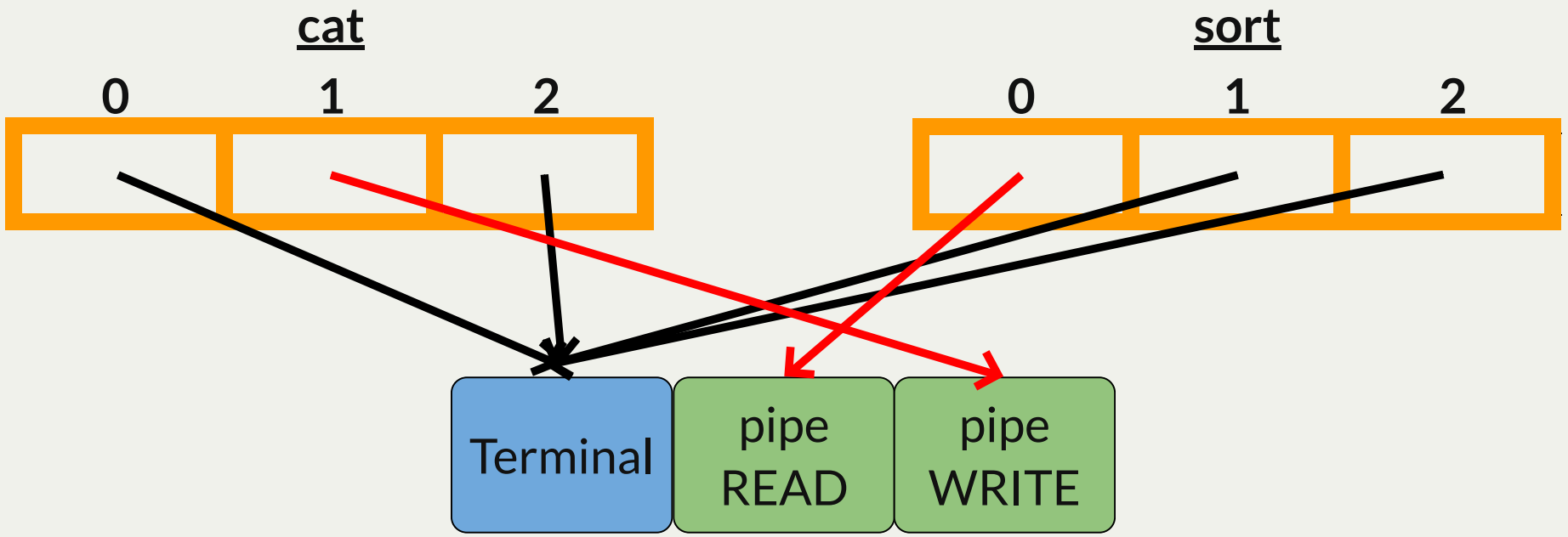
subprocess-soln.c

# Lecture Plan

- Review: pipes
- Redirecting process I/O
- ***Practice:*** Implementing **subprocess**
- ***Practice:*** **Implementing pipeline**

# Pipeline

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




**Final task:** write a program that spawns two child processes and connects the first child's STDOUT to the second child's STDIN.

# Redirecting Process I/O

Our final goal is to write a program that spawns two other processes where one's output is the other's input.  Both processes should run in parallel.

1. Our program creates a pipe
2. Our program spawns a child process
3. That child process changes its STDIN to be the pipe read end
4. That child process calls **execvp** to run the first specified command
5. Our program spawns another child process
6. That child process changes its STDOUT to be the pipe write end

# pipeline

Let's implement a custom function called **pipeline**.

```
void pipeline(char *argv1[], char *argv2[], pid_t pids[]);
```

**pipeline** is similar to **subprocess**, except it also spawns a second child and directs its STDOUT to write to the pipe.  Both children should run in parallel.

It doesn't return anything, but it writes the two children PIDs to the specified **pids** array

# Demo: pipeline

pipeline-soln.c

# pipe2

There were a lot of **close()** calls!  Is there a way for any of them to be done automatically?

```
int pipe2(int fds[], int flags);
```

**pipe2** is the same as **pipe** except it lets you customize the pipe with some optional flags.

- if flags is 0, it's the same as **pipe**
- if flags is **O_CLOEXEC**, the pipe FDs will *be automatically closed when the surrounding process calls* ***execvp***.

# pipeline

```
1  void pipeline(char *argv1[], char *argv2[], pid_t pids[]) {
2    int fds[2];
3    pipe(fds);
4
5    pids[0] = fork();
6    if (pids[0] == 0) {
7      close(fds[0]);
8      dup2(fds[1], STDOUT_FILENO);
9      close(fds[1]);
10     execvp(argv1[0], argv1);
11   }
12
13   close(fds[1]);
14
15   pids[1] = fork();
16   if (pids[1] == 0) {
17     dup2(fds[0], STDIN_FILENO);
18     close(fds[0]);
19     execvp(argv2[0], argv2);
20   }
21
22   close(fds[0]);
23 }
```

The highlighted calls to **close()** would no longer be necessary if we use **pipe2** with **O_CLOEXEC** because the surrounding process for each calls **execvp**.

Note that the parent must still close them because it doesn't call execvp.

# pipeline with pipe2

```
1  void pipeline(char *argv1[], char *argv2[], pid_t pids[]) {
2    int fds[2];
3    pipe2(fds, O_CLOEXEC);
4
5    pids[0] = fork();
6    if (pids[0] == 0) {
7      dup2(fds[1], STDOUT_FILENO);
8      execvp(argv1[0], argv1);
9    }
10
11   close(fds[1]);
12
13   pids[1] = fork();
14   if (pids[1] == 0) {
15     dup2(fds[0], STDIN_FILENO);
16     execvp(argv2[0], argv2);
17   }
18
19   close(fds[0]);
20 }
```

This version of pipeline uses **pipe2** with **O_CLOEXEC.**

# Pipes and I/O Redirection: Key Takeaways

- Pipes are sets of file descriptors that allow us to communicate across processes.
- Processes can share these file descriptors because they are copied on **fork()**
- File descriptors 0,1 and 2 are special and assumed to represent STDIN, STDOUT and STDERR
- If we change those file descriptors to point to other resources, we can redirect STDIN/STDOUT/STDERR to be something else without the program knowing!
- Pipes are how terminal support for piping and redirection (**command1 | command2** and **command1 > file.txt**) are implemented!

# Lecture Recap

- Review: pipes
- Redirecting process I/O
- ***Practice:*** Implementing **subprocess**
- ***Practice:*** Implementing **pipeline**

**Next time:** signals (another form of interprocess communication)

# 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

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 STDOUT.

```
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