CS111, Lecture 11
Pipes, Continued

Topic 2: Multiprocessing - How can our program create and interact with other programs? How does the operating system manage user programs?

CS111 Topic 2: Multiprocessing

assign3: implement your own shell!

Learning Goals

- Learn about **pipe** and **dup2** to create and manipulate file descriptors
- Use pipes to redirect process input and output

Plan For Today

- **Recap**: Pipes so far
- Redirecting Process I/O
- ***Practice:*** implementing **subprocess**
- ***Practice:*** implementing **pipeline**
- **pipe2**

```
cp -r /afs/ir/class/cs111/lecture-code/lect11 .
```

Plan For Today

- **Recap: Pipes so far**
- Redirecting Process I/O
- ***Practice:*** implementing **subprocess**
- ***Practice:*** implementing **pipeline**
- **pipe2**

```
cp -r /afs/ir/class/cs111/lecture-code/lect11 .
```

Pipes

- A pipe is (sort of) like an “imaginary file” that we open twice, once for writing and once for reading. It consists of two file descriptors, one for reading and one for writing
- Whatever we write to the “write FD” we can read from the “read FD”
- To create these two file descriptors, we call the **pipe** system call

```
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.

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

pipe() and fork()

- When we call **fork**(), everything is copied into the child, *including the file descriptors*. This means if a parent creates a pipe and then calls fork, the child can access it, too!
- However, this means both the parent *and* the child must close the pipe FDs when they are done with them.
  - Not closing them can cause functionality issues; for instance, if the child reads from the pipe until there is nothing left, but the write end of the pipe is not closed everywhere when we're done with it, the child will stall forever thinking more input could be coming.
- If someone tries calling **read** from a pipe and no data has been written, it will block until some data is available (or the pipe write end is closed).

pipe()

```
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;
}
```

Plan For Today

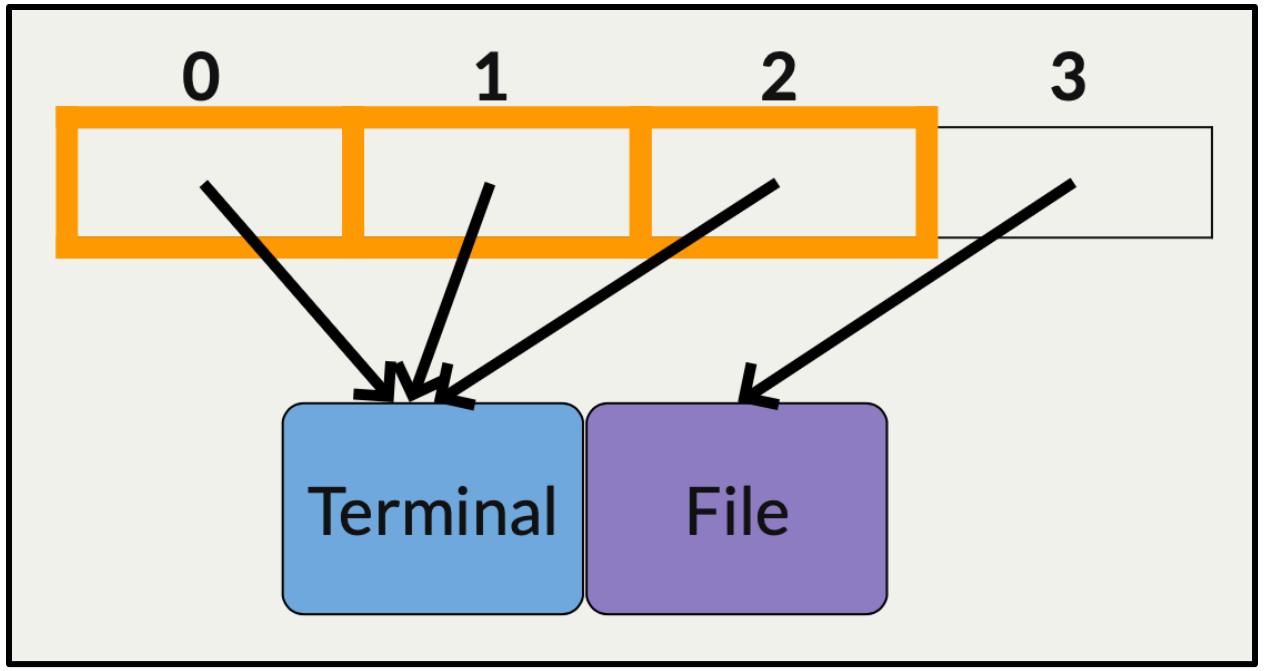
- Recap: Pipes so far
- **Redirecting Process I/O**
- ***Practice:*** implementing **subprocess**
- ***Practice:*** implementing **pipeline**
- **pipe2**

```
cp -r /afs/ir/class/cs111/lecture-code/lect11 .
```

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).

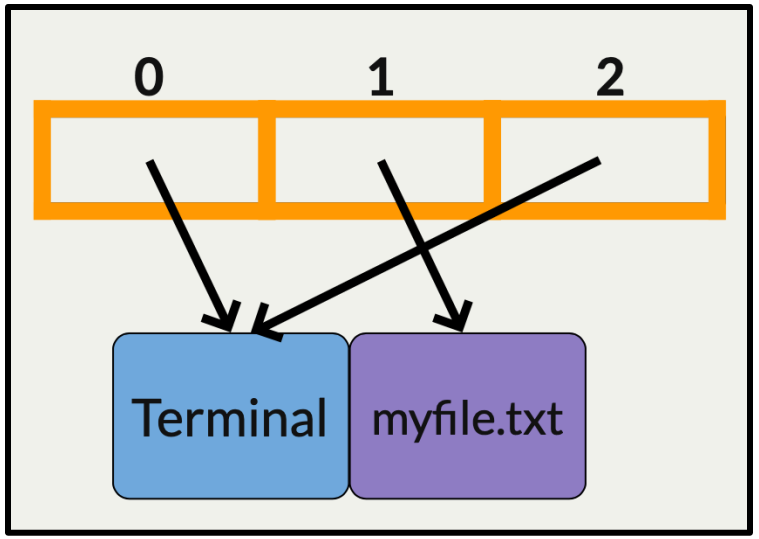


Idea: what happens if we change FD 1 to point somewhere else?

Redirecting Process I/O

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

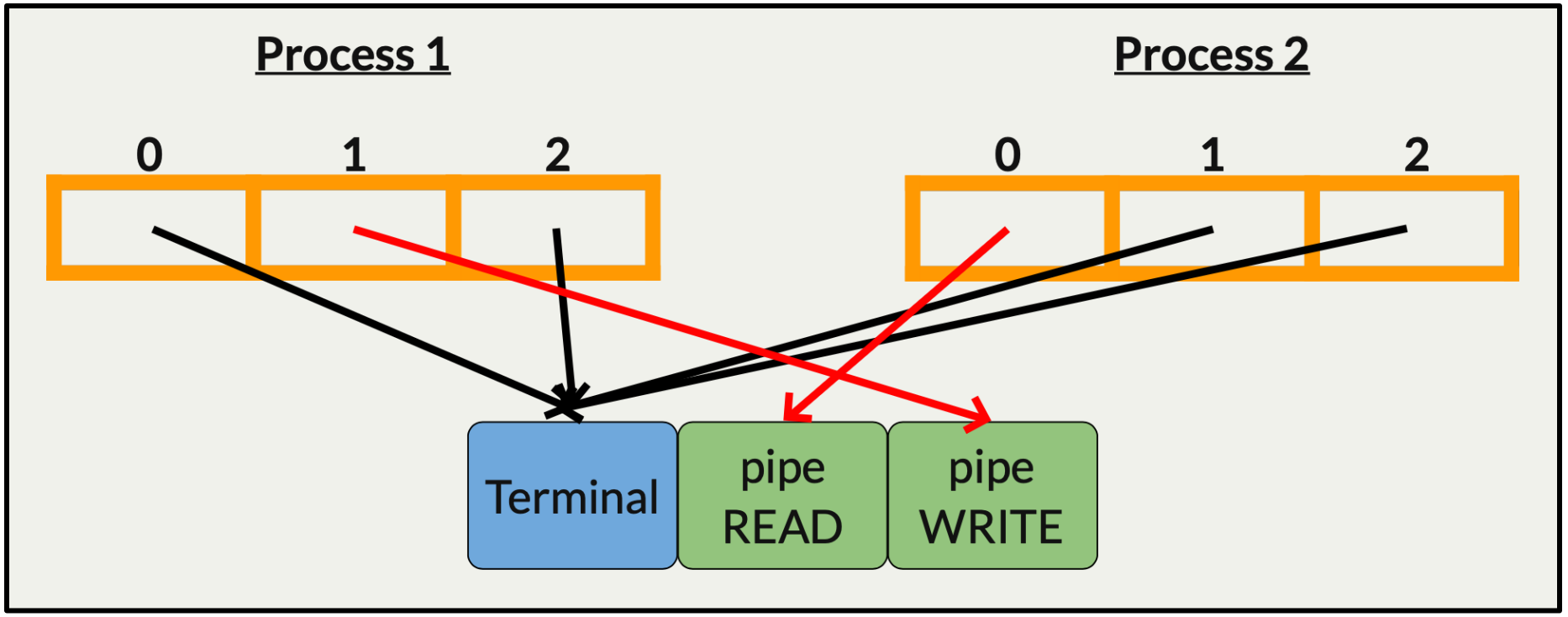
```
int main() {
    printf("This will print to the terminal\n");
    close(STDOUT_FILENO);

    // 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?

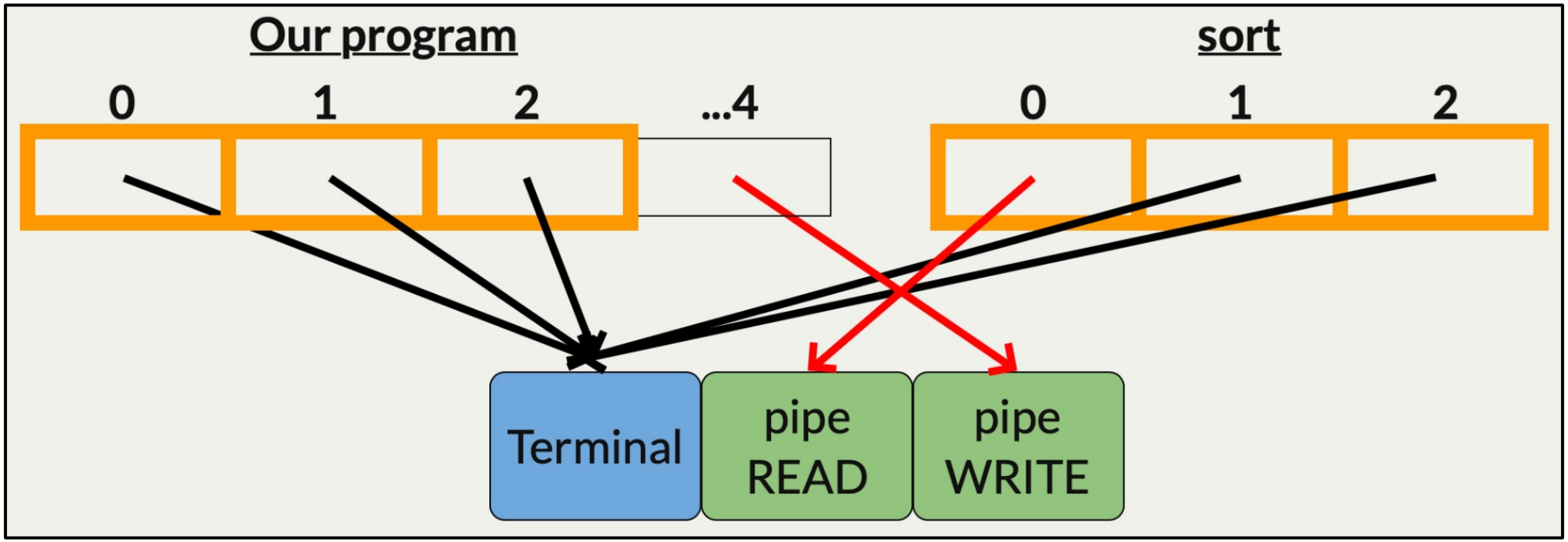
Could we do this with a pipe?



This is how piping works in the terminal! And the executables don't know they are using a pipe.

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

Stepping stone: 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?

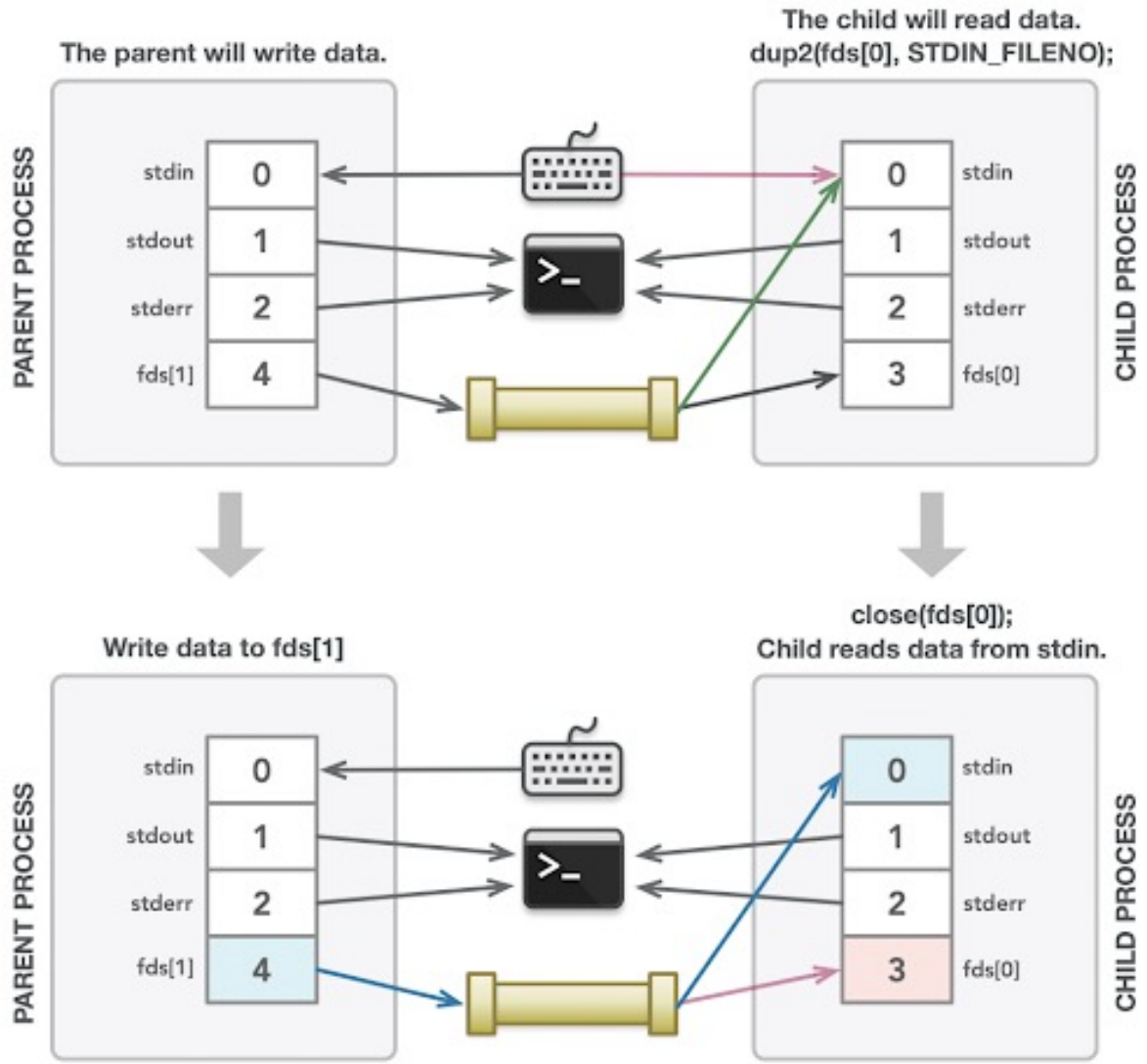
dup2 makes a copy of a file descriptor entry and puts it in another file descriptor index. If the second parameter is an already-open file descriptor, it is closed before being used.

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

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

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

Redirecting Process I/O



Illustrations courtesy of Roz Cyrus.

Plan For Today

- **Recap**: Pipes so far
- Redirecting Process I/O
- ***Practice:*** **implementing subprocess**
- ***Practice:*** implementing **pipeline**
- **pipe2**

```
cp -r /afs/ir/class/cs111/lecture-code/lect11 .
```

subprocess

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

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

subprocess is similar to our first shell example, 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

subprocess-soln.cc

How could we configure a process where we can read from its STDOUT via a pipe?

- Fork, then make a pipe, then connect write end to STDOUT
- Fork, then make a pipe, then connect read end to STDOUT
- Make a pipe, then fork, then connect write end to STDOUT
- Make a pipe, then fork, then connect read end to STDOUT

How could we configure a process where we can read from its STDOUT via a pipe?

Fork, then make a pipe, then connect write end to STDOUT

Fork, then make a pipe, then connect read end to STDOUT

Make a pipe, then fork, then connect write end to STDOUT

Make a pipe, then fork, then connect read end to STDOUT

How could we configure a process where we can read from its STDOUT via a pipe?

Fork, then make a pipe, then connect write end to STDOUT	
Fork, then make a pipe, then connect read end to STDOUT	
Make a pipe, then fork, then connect write end to STDOUT	✓ 0%
Make a pipe, then fork, then connect read end to STDOUT	

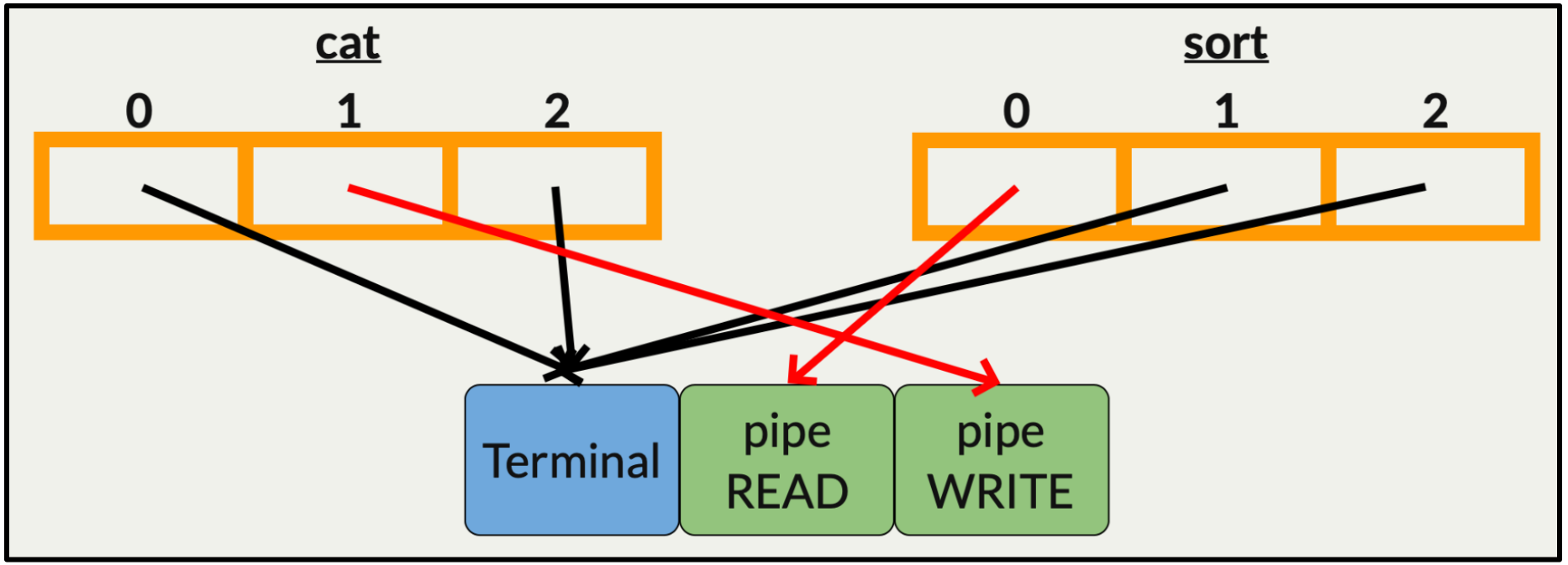
Plan For Today

- **Recap**: Pipes so far
- Redirecting Process I/O
- ***Practice:*** implementing **subprocess**
- ***Practice:*** **implementing pipeline**
- **pipe2**

```
cp -r /afs/ir/class/cs111/lecture-code/lect11 .
```

Pipeline

I/O redirection + pipes allow us to have 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.

Pipeline

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 **runTwoProcessPipeline**.

```
void runTwoProcessPipeline(const command& cmd1, const
                           command& cmd2, pid_t pids[]);
```

- **runTwoProcessPipeline** 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

pipeline-soln.cc

Plan For Today

- **Recap**: Pipes so far
- Redirecting Process I/O
- ***Practice:*** implementing **subprocess**
- ***Practice:*** implementing **pipeline**
- **pipe2**

```
cp -r /afs/ir/class/cs111/lecture-code/lect11 .
```

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**.*

pipe2

```
void runTwoProcessPipeline(const command& cmd1, const command& cmd2, pid_t
pids[]) {
  int fds[2];
  pipe(fds);

  // Spawn the first child
  pids[0] = fork();
  if (pids[0] == 0) {
    // The first child's STDOUT should be the write end of the pipe
    close(fds[0]);
    dup2(fds[1], STDOUT_FILENO);
    close(fds[1]);
    execvp(cmd1.argv[0], cmd1.argv);
  }

  // We no longer need the write end of the
  close(fds[1]);
  ...
```

The highlighted lines are not necessary if we use **pipe2** with **O_CLOEXEC** because the surrounding process calls **execvp**.

pipe2

```
  ...
  // Spawn the second child
  pids[1] = fork();
  if (pids[1] == 0) {
    // The second child's STDIN should be the read end of the pipe
    dup2(fds[0], STDIN_FILENO);
    close(fds[0]);
    execvp(cmd2.argv[0], cmd2.argv);
  }

  // We no longer need the read end of the pipe
  close(fds[0]);
}
```

The highlighted line is not necessary if we use **pipe2** with **O_CLOEXEC** because the surrounding process calls **execvp**.

Pipes and I/O Redirection

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!

assign3

Implement your own shell! (“stsh” – Stanford Shell)

4 key features:

- Run a single command and wait for it to finish
- Run 2 commands connected via a pipe
- Run an arbitrary number of commands connected via pipes
- Have command input come from a file, or save command output to a file

Recap

- **Recap**: Pipes so far
- Redirecting Process I/O
- ***Practice:*** implementing **subprocess**
- ***Practice:*** implementing **pipeline**
- **pipe2**

Lecture 11 takeaway: We can share pipes with child processes and change FDs 0-2 to connect processes and redirect their I/O.

Next time: introduction to multithreading

```
cp -r /afs/ir/class/cs111/lecture-code/lect11 .
```