Pipes

- Topics
  - Pipes: What are they?
  - How do you set them up?
- Learning Objectives:
  - Identify when file descriptors share entries in the file open table and when they do not.
  - Be able to write code to properly set up a pipe between different kinds of processes:
    - Parent/child
    - Siblings

Communicating among processes

- You've all used the | character to create pipes on the command line in the shell (I hope).
- And you should have all worked on optimizing a pipe implementation for Weensy.
- But, what exactly is a pipe?
- The effect:
  - When you type:
    ```
    % foo | bar
    ```
  - The `stdout` stream of foo is connected to the `stdin` stream of bar.
- You've probably used the file pointers, `stderr`, `stdout`, `stdin` in `fprintf` and `fscanf`.
- STDIN_FILENO/STDOUT_FILENO(and STDERR_FILENO) are the corresponding file descriptors.
- They are opened on behalf of every process.
  - By convention, `stdin` comes from the console
  - By convention, `stdout` and `stderr` go to the display
- Allowing two proceses to interact as shown above requires that we connect foo's `stdout` to bar's `stdin`

File Descriptors and `fork`

- Recall that when a parent forks:
  - **Any open files in the parent are open in the child.**
  - **The parent and child share the open-file structure referenced by the file descriptors**
  - **The parent and child share the same offset**
- Let's look at `share.c` and `own.c`
  - `share.c` opens a file before forking
  - `own.c` opens a file after forking
  - Run each and examine the output in `data.out`

Creating a pipe: The `pipe` system call

- `pipe(int filedes[2])` creates a pair of file descriptors and places them in the array referenced by filedes.
  - filedes[0] is for reading
  - filedes[1] is for writing
- By combining, `fork`, `exec`, and `pipe`, parents can communicate with children and/or set up pipelines between children.

Screen Capture

- Let's look at `pipe.c`
- Now, let's run it.
  - Notice: The child is the only one printing, and it can only print what it reads from the pipe, so anything the parent writes on the pipe gets read by the child.
  - Notice: After the parent exits, the child is still running. Type `ps` to see this. (You can kill it by typing `kill pid`. E.g., `kill 4932`).
  - Why?????

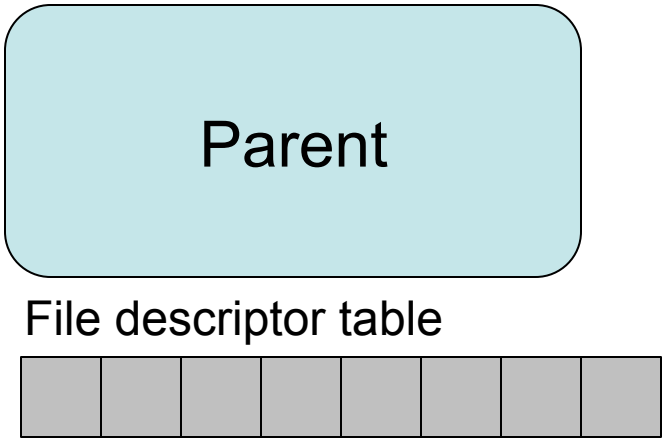
Pipe Hygiene?

- This is the term we use to describe the process of properly closing any unused ends of pipes.
- Why do we care?
  1. From the man page, “The pipe itself persists until all of its associated descriptors are closed.”
     - Implication if we don’t close descriptors, then the pipe could stick around a long time/forever. Pipes consume operating system resources, so you probably don’t want that.
  2. Let’s say that we have Parent writing into a pipe from which Child is reading.
     - When will Child get EOF on the pipe?
     - Answer: when the write end is closed.
     - Implication: if we create a pipe between between two processes and only one ever closes the write end, the other could block forever trying to read from the pipe, because it will never receive the eof indicator.

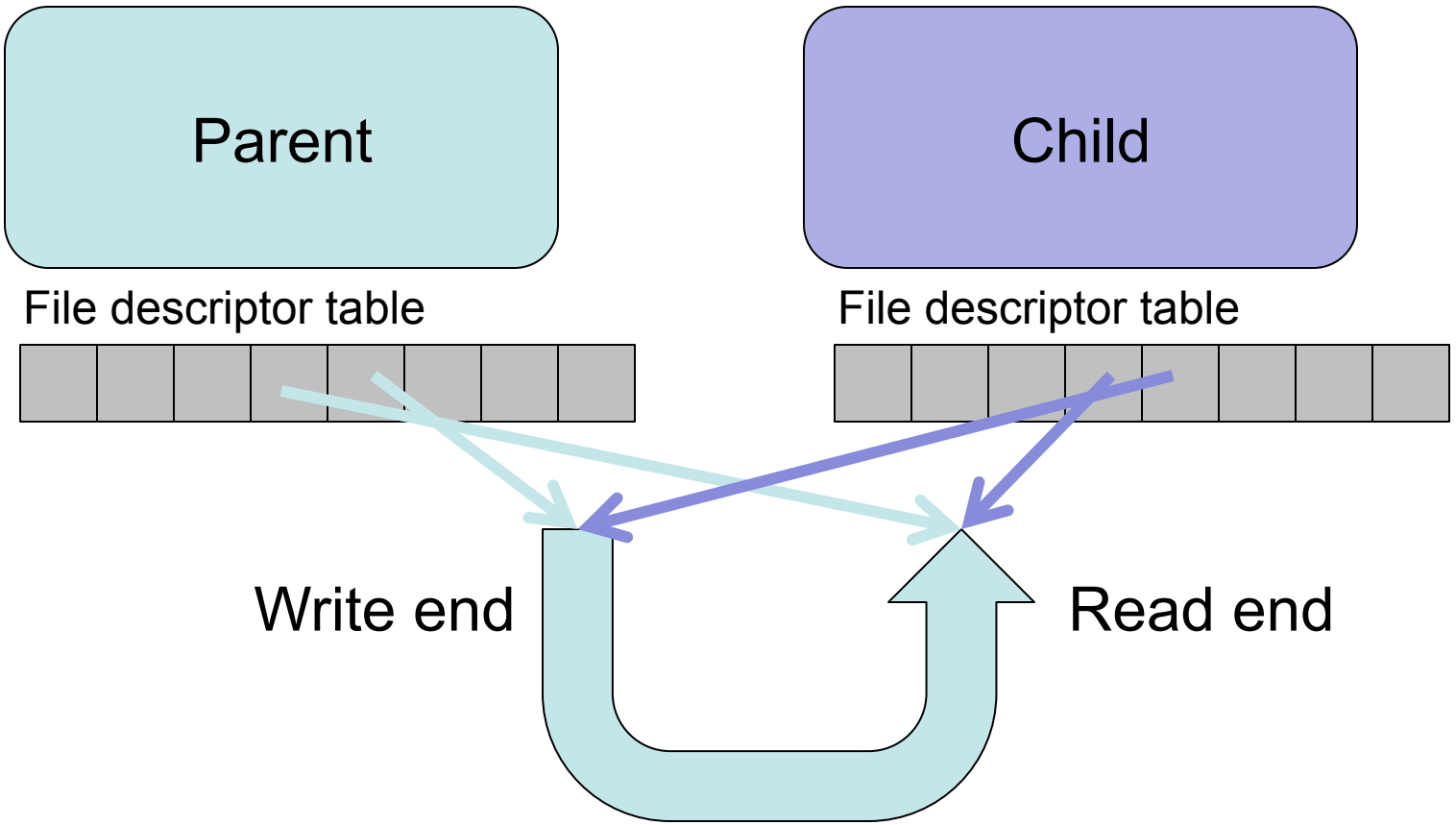
Pipe Specifications

- A pipe whose read or write end is closed (by all opens) is called widowed.
- The only way to deliver EOF is to widow the pipe (i.e., close the write end).
  - At that point, a reader will read any data that has been buffered from the writer and after consuming all that data will get a 0-byte return.
- If you try to write to a widowed pipe (i.e., the read end is closed) the writing process will get a SIGPIPE signal.

Pipes by Pictures



Pipes by Pictures



Screen Capture

- Let's add a close call where we think we need it.
- The file `pipe1.c` contains a version with this change in it.
- Now, if we run pipe, does the child exit?

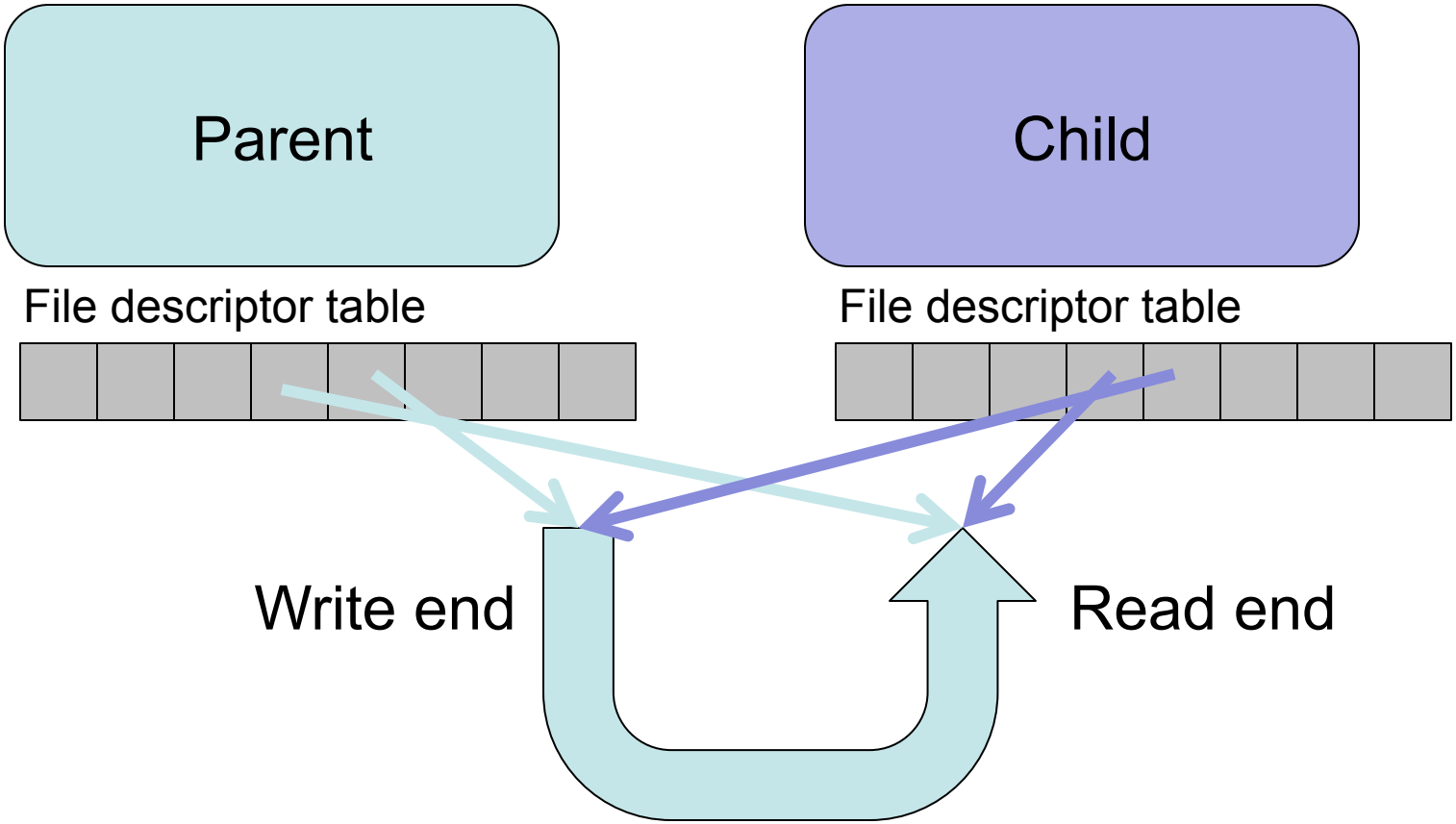
Pipes and `stdin/stdout`

- So, we've created a pipe and we can write into it and have another process read from it.
- How do we make this work for `stdin` and `stdout`?

The `dup2` system call

- `dup2(int filedes, int filedes2)`
  - duplicates the first file descriptor (`filedes`) into the second file descriptor (`filedes2`).
  - After the call, both file descriptors refer to the same object, so reading from/writing to one descriptor changes the file position in both descriptors.
  - If `filedes2` already refers to an open object, that object is closed.
  - How does this help us?

Pipes by Pictures



Screen Capture

- Let's use dup2 and see if we can write to `stdout` and read from `stdin`.
- The code that does this is in `pipe2.c`
- The code in `pipe3.c` uses `printf` and `scanf` to really convince you that this is what's happening.

Creating a Pipeline (foo | bar)

Note: Terrible error handling to save space!

```
pid_t child1, child2;
int pipedes[2], status;

assert (pipe(pipedes) == 0);                /* Create the pipe. */
child1 = fork();
if (child1 == 0) {
           /* child */
           close(pipedes[0]);               /* Close read end */
           dup2(pipedes[1], STDOUT_FILENO); /* Make stdout the same as the pipe write fd */
           execvp("foo", argv);             /* Assume argp is set */
}
/* only parent gets here */
child2 = fork();
if (child2 == 0) {
           /* child */
           close (pipedes[1]);              /* Close writing end */
           dup2(pipedes[0], STDIN_FILENO);  /* Make stdin the same as the pipe read fd */
           execvp("bar", argv);
}
/* Parent once again */
close (pipedes[0]);                         /* Close pipe fDs in parent. */
close (pipedes[1]);
waitpid(child2, &status, 0);                /* Wait for second process to complete. */
```

Picture of Pipeline