Pipes

TABLE OF CONTENTS

Task

Other times, it is useful to provide the output of a program as the input of another program. The syntax `[process A] |[process B]` tells your shell to pipe the output of process A to the input of process B. In other words, the output of program A becomes the input of program B.

Modify your shell so that it supports pipes between programs. You can assume there will always be spaces around the special character `|`.

You must support an indefinite number of pipes in series, as seen in the example below.

```
[process A] | [process B] | [process C]
```

To implement this part of the shell, you will need to use the `pipe` syscall. An explanation for how this syscall works is provided below. Historically, this portion of the homework has been the most difficult for students to implement due to students not fully understanding what's actually going on when they create pipes. Thus, we **highly recommend** that you first fully understand how pipes work on a conceptual level before you begin coding. **Course staff reserves the right to not help you debug your code if you prove that you started coding without first attempting to understand the fundamental concepts outlined below.**

Background

A pipe is a mechanism for inter-process communication, where one process writes to the “write end” of the pipe, and another process reads from the “read end”. Understanding the implementation details of pipes is not required for this homework.

How the `pipe` syscall works

To create a pipe in code, we use the `pipe` syscall: `int pipe(int fds[2])`.

As a bit of review, when a process opens a regular file (using the `open` syscall), a file description is created and it is assigned a file descriptor (remember, this is just a non-negative integer). This mapping between file descriptor and description is stored in the process’s FDT.

On the other hand, when a process opens a pipe (using the `pipe` syscall), two file descriptions are created: one for the read end and one for the write end. After calling `pipe`, `fds[0]` gets set to the file descriptor for the read end of the pipe, and `fds[1]` gets set to that of the write end.

Just as the `read` and `write` syscalls operate on file descriptors in the “regular file” case, `read` and `write` are used to interact with the file descriptors corresponding to the read and write ends of the pipe respectively.

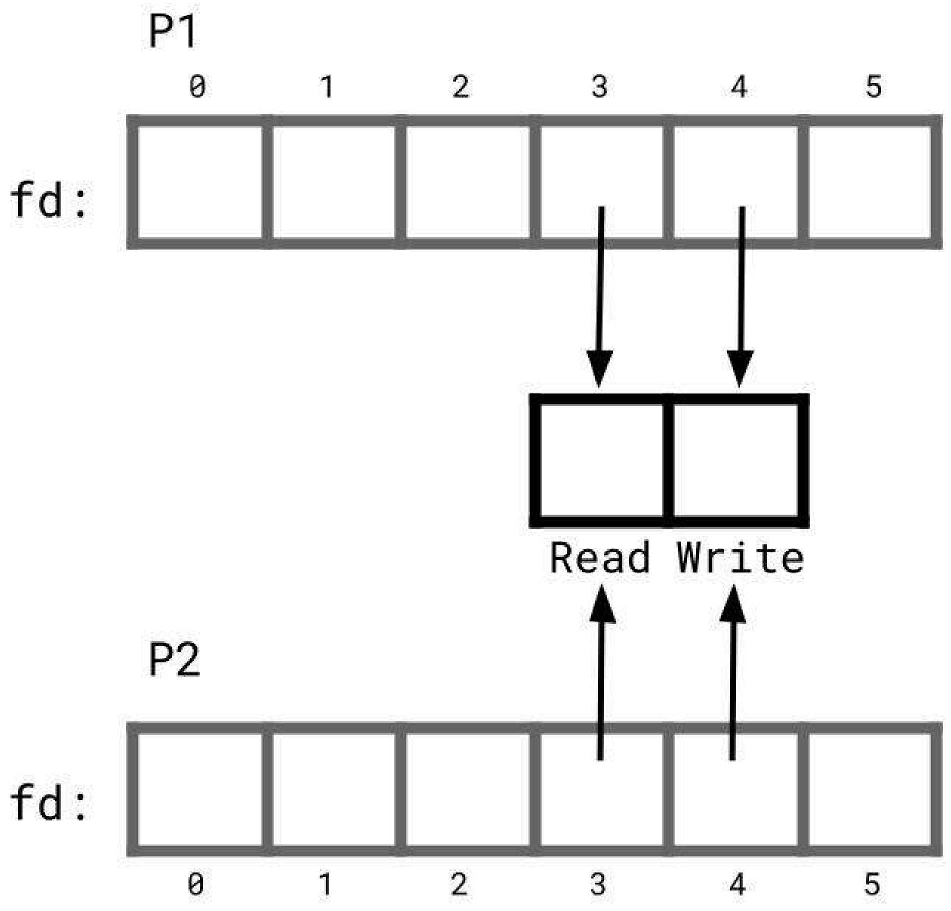
When a process tries to read from the read end of the pipe, it will block until data shows up at the write end (i.e. another process writes to the write end). When all file descriptors pointing to the write end of the pipe have been closed, calling `read` on the read end of the pipe will return `EOF` (i.e. 0).

Pipes have limited capacity, meaning they can only store so much data. When a process tries to write to the write end of the pipe and the pipe is full (i.e. another process hasn’t read from the read end to “empty out the pipe”), the writing process will block. When all file descriptors pointing to the read end of the pipe have been closed, writing to the pipe will raise the `SIGPIPE` signal.

Walkthrough of a simple pipe example

Let’s say that we want a parent process P1 to write data to a pipe and have its child process P2 read the data.

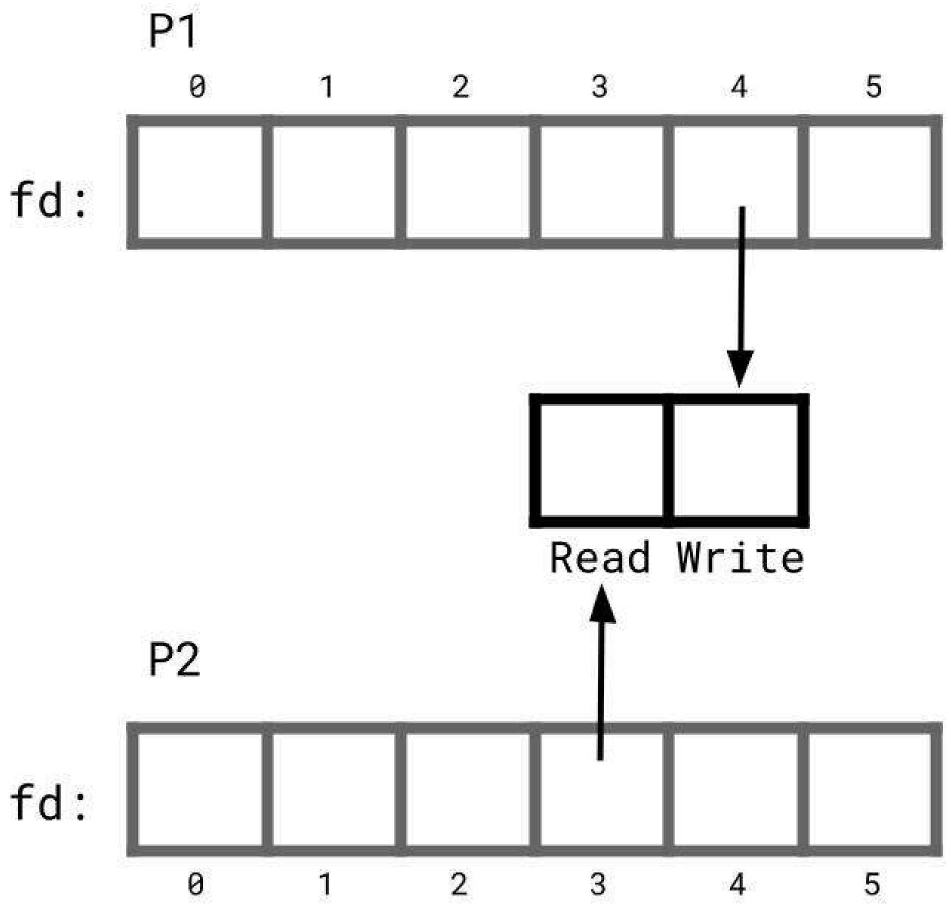
P1 first creates a pipe and then creates P2 using `fork()`. The diagram below depicts the FDTs of P1 and P2, assuming that the `pipe` syscall set FDs 3 and 4 to point to the read and write end of the pipe respectively. Remember that because P2 was created using `fork()`, P1 and P2 have identical FDTs that point to the same file descriptions.



P1 calls `close(3)` and P2 calls `close(4)`. Now, P1 only has an open file descriptor for the write end of the pipe, and P2 only has an open descriptor for the read end. P1 can happily write data to the write end and P2 will continue to read this data until either

1. P1 chooses to stop sending data, in which case P1 would simply call `close(4)`.
2. P2 chooses to stop reading from the pipe, in which case P2 would simply call `close(3)`.

In case 1, after P1 closes its write-end FD, P2 will be notified that there’s nothing left to read (the `read` syscall will return 0 bytes read), so it won’t block indefinitely. In case 2, after P2 closes its read-end FD, P1 will be notified that there’s no one left to read the data (via the `SIGPIPE` signal, which P1 can choose to handle gracefully so as to not exit), so it also won’t block.



The steps above where the writer process closes its read-end FD and the reader process closes its write-end FD are extremely important. Let's consider the importance of each step separately.

Recall that a process that reads from a pipe will continue to expect data to show up at the write end as long as there are open file descriptors pointing to the write end. Thus, if P2 doesn't call `close(4)`, even when P1 is done sending its data through the pipe and closes its own write-end FD, P2 will continue to hang, waiting for data that will never come.

Similarly, consider the case where P1 doesn't call `close(3)`. Let's assume that in this scenario, P1's job is to keep writing data to the pipe until there are no longer any processes reading from the pipe. Consider what happens when P2 finishes execution at some point in time. When P2 exits, all of its FDs get closed. However, there remains an open FD for the read end of the pipe in P1's FDT, so P1 thinks that there's still a process that's reading data from the pipe. P1 continues writing data to the pipe and eventually, the pipe fills up completely, since there's no longer a reader to "empty it out". When P1 attempts to write to the full pipe, it will get blocked indefinitely.

Recommended approach for implementing piping

Let's say that the user inputs a command that involves creating `n` processes, where process `i` treats the output of process `i - 1` as its `stdin` and process `i` sends its `stdout` into process `i + 1`'s `stdin`. Having `n` processes means we need `n - 1` pipes.

1. Create an array of `n - 1` pipes in the shell process. Let's call this array `pipe_arr`.
2. Use a for loop to `fork()` the `n` child processes.
3. For every index `i` (aside from the first and last, which are edge cases), process `i` changes its `stdin` FD (0) to point to the read end of `pipe_arr[i - 1]` and changes its `stdout` FD (1) to point to the write end of `pipe_arr[i]`. The `dup2` syscall will prove to be useful here. The idea here is that for every index `i` (other than the edge cases), process `i` is responsible for writing to the `i`th pipe in `pipe_arr`. Because every process does this, process `i` can read from the read end of `pipe_arr[i - 1]` to get data that's being written by process `i - 1`.
4. After changing its `stdin` and `stdout`, process `i` **must close all of its pipe FDs**. If these FDs are not closed, **the process will hang** for the reasons provided in the previous section. After closing all of these pipe FDs, the process will still have access to a single read-end FD and a single write-end FD for the appropriate pipes due to using `dup2`.
5. The shell process also needs to close all of the pipe FDs since it doesn't make use of them at all (remember, the pipes are used exclusively by the child processes).

Hopefully, these guidelines allow you to implement piping in a relatively hassle-free manner. Remember, if you ever run into issues with your code hanging, there's a very high chance that you didn't close all of the necessary pipe FDs.