Exploration: Pipes & FIFO

Introduction

In a previous exploration, we had used the `dup2()` function to redirect standard input and standard output of a process. But what if two processes wanted to send each other data? We could set up a data channel between them by using an intermediate file, and then redirecting the sender/writer process's standard output to write to this file and redirecting the receiver/reader process's standard input to read from that file. Pipes and FIFOs provide us simple mechanisms for achieving such communication between processes.

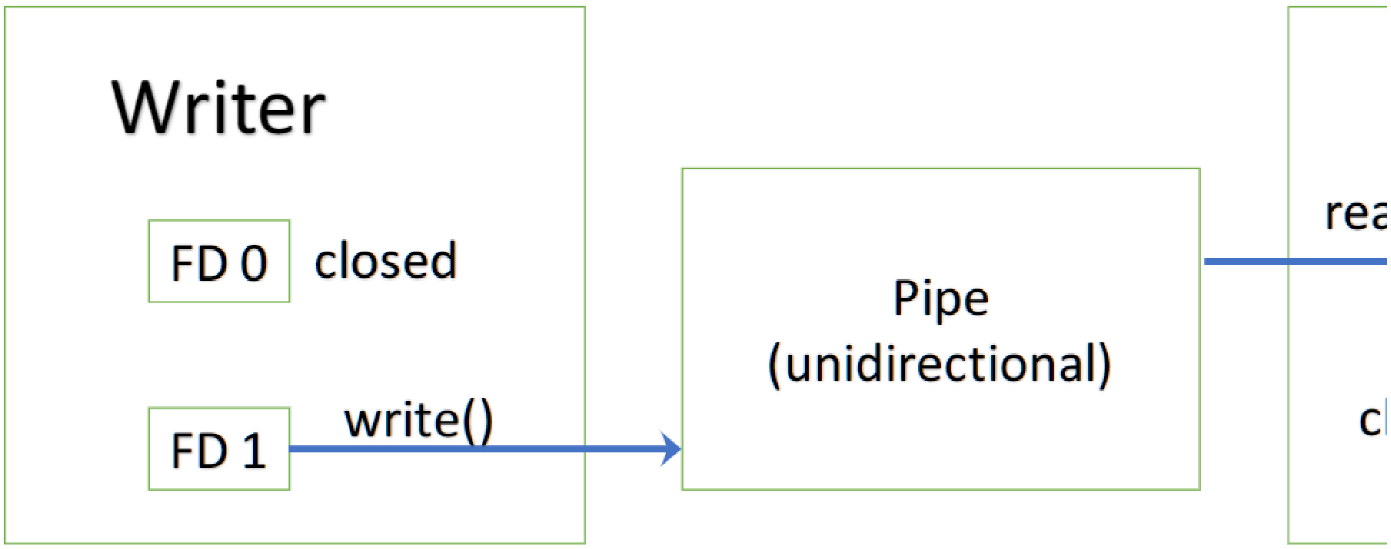
Pipes

Pipes are used for Inter-process communication (IPC) between two processes that are forked by a common ancestor process. A pipe is a **unidirectional data channel** that connects a write-only file descriptor in one process to a read-only file descriptor in another process.

Creating a Pipe

Pipes are possible because file descriptors are shared across `fork()` and `exec()` family of functions. Here is how pipes are created and used:

- A parent process creates a pipe by calling the `pipe()` function. This results in two new open file descriptors, one for input and one for output.
- The parent process then calls `fork()` and possibly `exec()`. Now both the parent and the child have the two file descriptors that were created by the `pipe()` function.
- If a pipe is to used for the parent process to send data to the child process, then the parent process writes to the output file descriptor and the child process reads from the input file descriptor.
- Conversely, if a pipe is to be used for the child process to send data to the parent process, then the child process writes to the output file descriptor and the parent process reads from the input file descriptor.



Using a pipe to send data from the writer process to the reader process

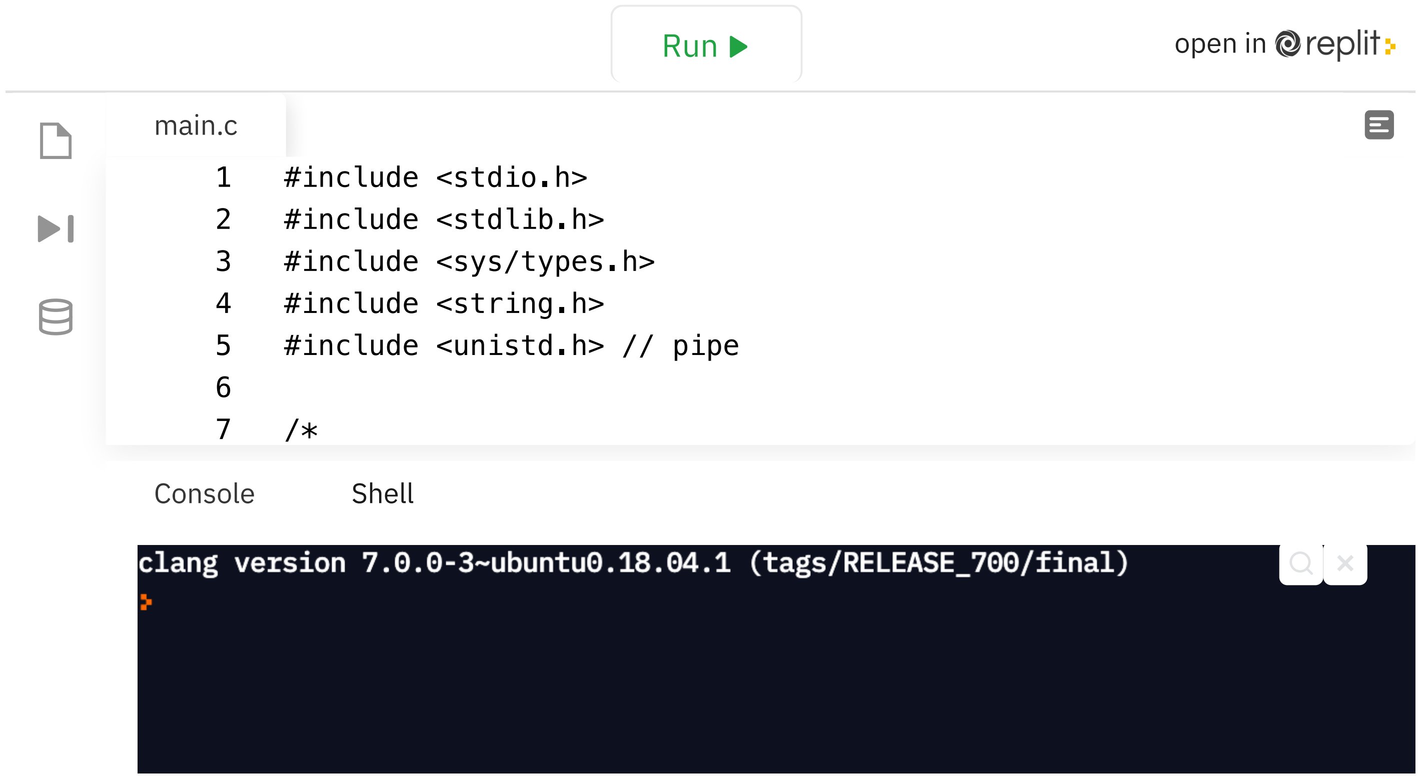
The `pipe()` function

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

The `pipe` **function** **(https://man7.org/linux/man-pages/man2/pipe.2.html)** takes one argument whose data type is an array of two file descriptors. The function returns two file descriptors in this ay. The file descriptor in `pipefd[0]` refers to the read end of the pipe, and the file descriptor in `pipefd[1]` refers to the write end of the pipe. The kernel buffers the data written to the write end of the file until the process at the read end of the pipe reads it. On success, the function returns the value 0. On error, the function returns the value –1.

Example: Using Pipes

In the following program, we create a pipe that is used by the child process to send data to the parent process. The child process uses the `write()` system call (which we discussed in Module 5) to write data to the pipe. The parent process, which is the reader, uses `read()` system call to read data from the pipe. When data is available `read()` immediately returns the data. The return value of `read()` tells you how many bytes were read which can be less than how many you requested. If there is no data available in the pipe, the call to `read()` is blocked until data becomes available.



```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <string.h>
#include <unistd.h> // pipe

/*
```

```
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)
```

Note that pipes have a certain size. On Linux this size is typically 65,536 Bytes, i.e., 64 KBytes. If a pipe fills up and there is no more room, then `write()` will block until space becomes available which will happen when the reader process reads the data from the pipe.

The read() function

```
#include <unistd.h>
ize_t read(int fd, void *buf, size_t count);
```

The `read()` function reads data from the file descriptor `fd`. It attempts to read up to `count` bytes into the buffer pointing at by `buf`. On success, `read()` returns the number of bytes it read. This number can be less than `count` if, for example, there are fewer bytes available to read, hence this does not indicate an error. `read()` returns –1 on error and returns 0 on encountering end-of-file.

The strstr() function

In the program we use **the string function `strstr()` (https://man7.org/linux/man-pages/man3/strstr.3.html)** to find the termination indicator, @@, inside the data that is read in. Let's look at this function:

```
#include <string.h>
char *strstr(const char *haystack, const char *needle);
```

The `strstr()` function finds the first occurrence of the substring `needle` in the string `haystack`. If the substring is found then the function returns a pointer to the beginning of the located substring. If the substring is not found then it returns NULL.

Closing Pipes

What happens if one of the processes closes its end of the pipe?

- If the reader process closes the input pipe, then when the writer process tries to write to the pipe, `write()` will return –1. Additionally, the writer process will be sent the SIGPIPE signal which we described in the exploration related to signals.
- If the writer process closes the output pipe, then the process reading from the pipe will read end-of-file after it has read all remaining data in the pipe and will return 0.

FIFO

FIFO stands for First-in, First-out. FIFO are very similar to pipes with the crucial difference that a FIFO has a name. This allows FIFOs to be used for communication between two process on the same machine even if these processes are not related to each other. Because FIFOs have names and they are similar to pipes, they are also called **named pipes**.

A FIFO is essentially a persistent pipe, which is represented by a special file. We create a FIFO by calling the function `mkfifo()`. Once a FIFO is created any process can open a FIFO with `open()` and then use it just like using a pipe. When opening a FIFO, `open()` called by the first process will block. The first process will unblock once the second process also calls `open()`

The mkfifo()function

```
#include <sys/types.h>
#include <sys/stat.h>

t mkfifo(const char *pathname, mode_t mode);
```

This function (https://man7.org/linux/man-pages/man3/mkfifo.3.html) creates a FIFO with the given `pathname` and the permissions specified by `mode`.

Example: FIFO

In the following example, there are two programs, a reader program and a writer program. The programs use @@ as a terminal indicator, i.e., to indicate that the message is complete.

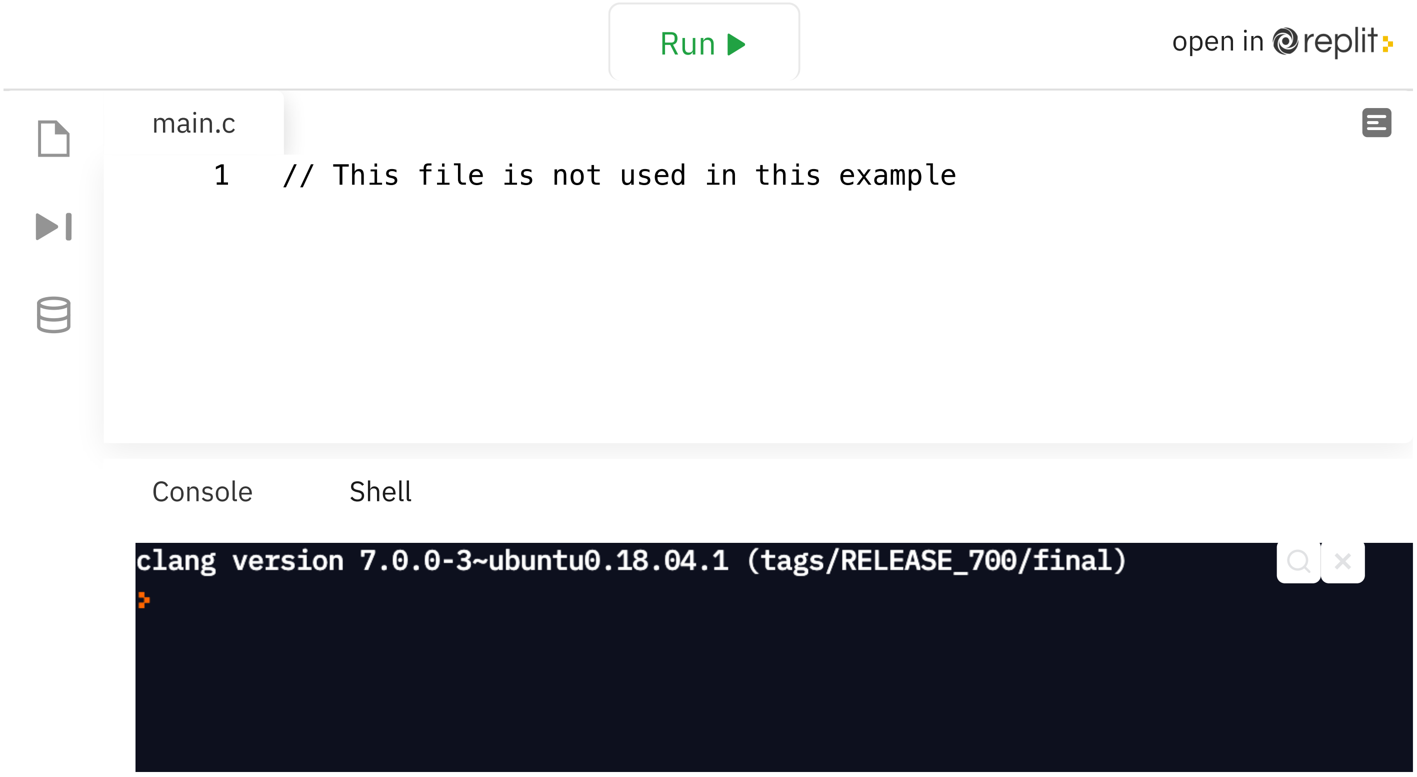
Note: Click on the "File" icon on the left side of the repl to view a list of files. The file 7_1_reader.c is the reader program, and the file 7_1_writer.c is the writer program.

The reader program

- Creates a FIFO by calling `mkfifo()`
- Opens the FIFO by calling `open()`
- Reads from the FIFO by calling `read()`
- When it finds the terminal indicator in the data, the reader program prints the complete message to standard output and exits.

The writer program

- Opens the FIFO by calling `open()`
- Writes to the FIFO by calling `write()` and exits.
- A small sleep has been added in the beginning of the writer to make sure that the FIFO has been created by the reader before the writer tries to open it.



```
// This file is not used in this example
```

```
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)
```

eating FIFOs from the shell

A FIFO can also be created from the shell, by using **the command `mkfifo` (https://man7.org/linux/man-pages/man1/mkfifo.1.html)**. Since a FIFO is a file, it will be listed by the `ls` command. For example, the following command will create a FIFO named `my_fifo` in the current directory, which is listed by the `ls` command as a special file indicated by the character `p` in the first column.

```
$ mkfifo my_fifo
$ ls -l my_fifo
prw-rw----. 1 chaudhrn upg11000 0 May 25 06:13 my_fifo
```

Additional Resources

Here are some references to learn more about the topics we discussed in this exploration.

- Pipes and FIFOs are discussed in Chapter 44 of *The Linux Programming Interface*.

  Kerrisk, M. (2010). *The Linux programming interface : a Linux and UNIX system programming handbook*. San Francisco: No Starch Press.