STREAMS message/PIPEs/FIFO:pipe, popen and pcloseFunctions

Assignment No: 13_a

Subject:- Unix Operating System

System Lab Class :- TYIT

Name PRN

Aditi Sudhit Ghatle 2020BTEIT00044

Title:

Send data from parent to child over a pipe

Objectives:

1. To learn about STREAMS message/PIPEs/FIFO:pipe, popenand pcloseFunctions

Theory:

Pipes

Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems. Pipes have two limitations:

1. Historically, they have been half duplex (data flows in only one direction). Some systems now provide full-duplex pipes, but for maximum portability, we should never assume that this is the case.
2. Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child.

FIFOs (Section 15.5) get around the second limitation, and that UNIX domain sockets (Section 17.2) get around both limitations.

Despite these limitations, half-duplex pipes are still the most commonly used form of IPC. Every time you type a sequence of commands in a pipeline for the shell to execute, the shell creates a separate process for each command and links the standard output of one process to the standard input of the next using a pipe.

A pipe is created by calling the `pipe` function.

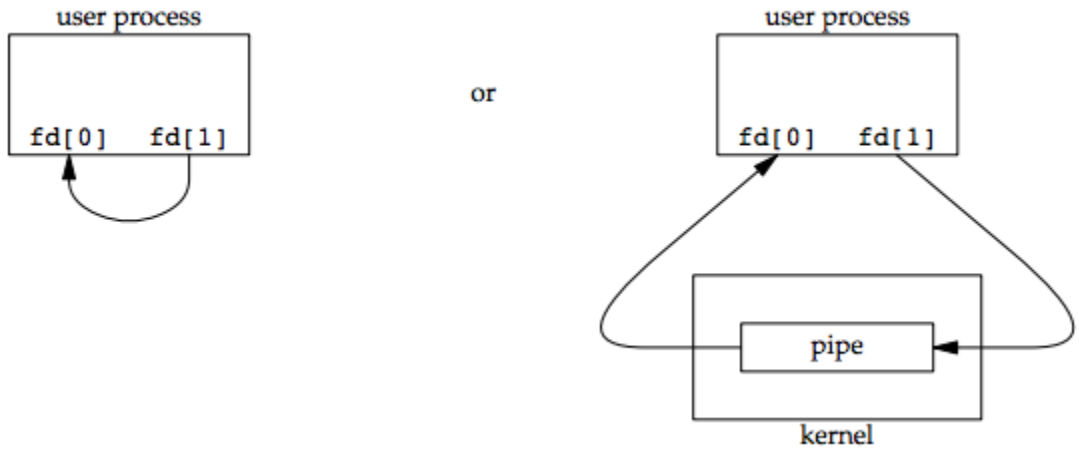
```
#include <unistd.h>


int pipe(int fd[2]);
```

```
/* Returns: 0 if OK, −1 on error */
```

Two file descriptors are returned through the *fd* argument: *fd[0]* is open for reading, and *fd[1]* is open for writing. The output of *fd[1]* is the input for *fd[0]*.

POSIX.1 allows for implementations to support full-duplex pipes. For these implementations, *fd[0]* and *fd[1]* are open for both reading and writing.

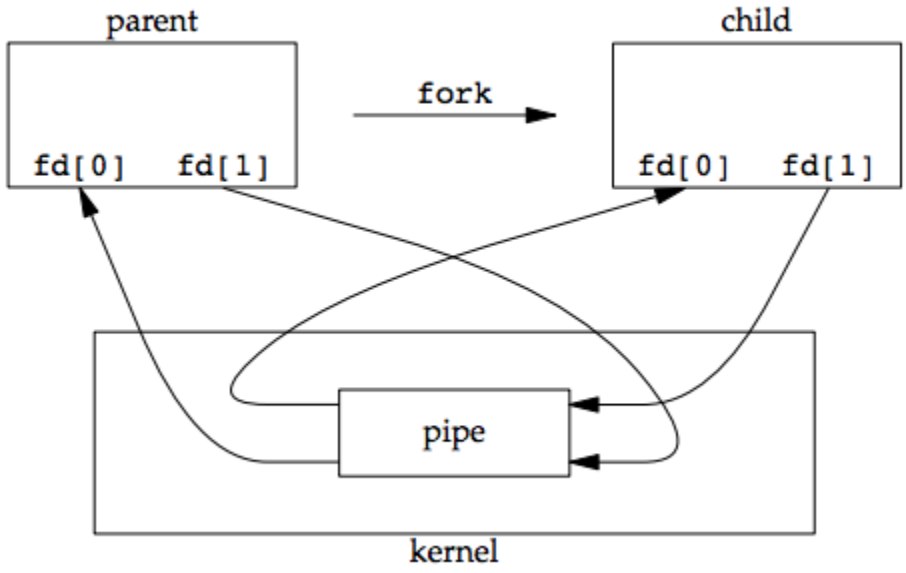
Two ways to picture a half-duplex pipe are shown in the figure below. The left half of the figure shows the two ends of the pipe connected in a single process. The right half of the figure emphasizes that the data in the pipe flows through the kernel.



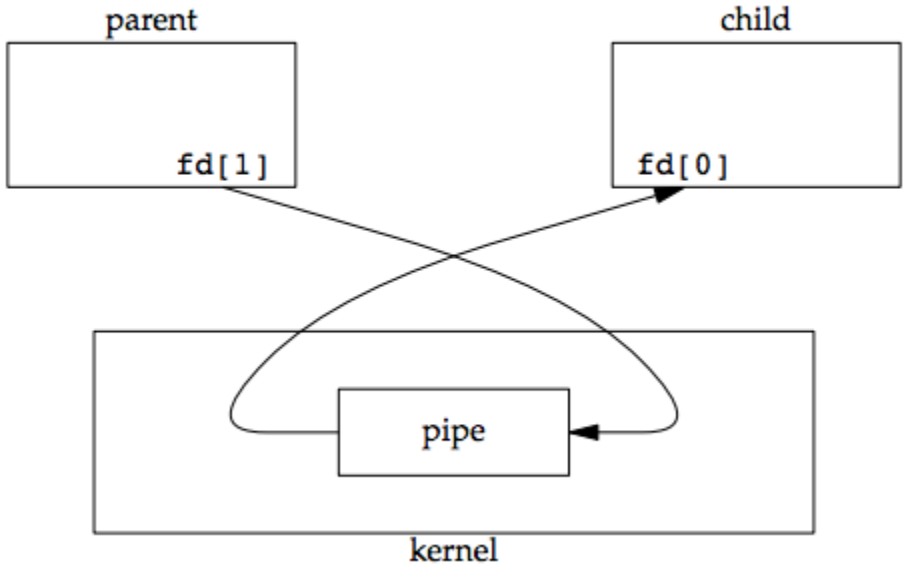
The `fstat` function returns a file type of FIFO for the file descriptor of either end of a pipe. We can test for a pipe with the `S_ISFIFO` macro.

POSIX.1 states that the `st_size` member of the stat structure is undefined for pipes. But when the `fstat` function is applied to the file descriptor for the read end of the pipe, many systems store in `st_size` the number of bytes available for reading in the pipe, which is nonportable.

A pipe in a single process is next to useless. Normally, the process that calls `pipe` then calls `fork`, creating an IPC channel from the parent to the child, or vice versa. The following figure shows this scenario:



What happens after the `fork` depends on which direction of data flow we want. For a pipe from the parent to the child, the parent closes the read end of the pipe (*fd[0]*), and the child closes the write end (*fd[1]*). The following figure shows the resulting arrangement of descriptors.



For a pipe from the child to the parent, the parent closes fd[1], and the child closes fd[0].

When one end of a pipe is closed, two rules apply:

1. If we read from a pipe whose write end has been closed, read returns 0 to indicate an end of file after all the data has been read.
    - Technically, we should say that this end of file is not generated until there are no more writers for the pipe.
    - It's possible to duplicate a pipe descriptor so that multiple processes have the pipe open for writing.
    - Normally, there is a single reader and a single writer for a pipe. (The FIFOs in the next section dicusses that there are multiple writers for a single FIFO.)

2. If we write to a pipe whose read end has been closed, the signal SIGPIPE is generated. If we either ignore the signal or catch it and return from the signal handler, write returns −1 with errno set to EPIPE.

When we're writing to a pipe (or FIFO), the constant `PIPE_BUF` specifies the kernel's pipe buffer size. A write of `PIPE_BUF` bytes or less will not be interleaved with the writes from other processes to the same pipe (or FIFO). But if multiple processes are writing to a pipe (or FIFO), and if we write more than `PIPE_BUF` bytes, the data might be interleaved with the data from the other writers. We can determine the value of `PIPE_BUF` by using `pathconf` or `fpathconf`.

Example: creating a pipe between a parent and its child

ipc1/pipe1.c

```
#include "apue.h"

int
main(void)
{
    int     n;
    int     fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {       /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else {                    /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
```

```
    exit(0);
}
```

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
        int fd[2];
        int childID = 0;

        // create pipe descriptors
        pipe(fd);

        // fork() returns 0 for child process, child-pid for parent process.
        if (fork() != 0) {
                // parent: writing only, so close read-descriptor.
                close(fd[0]);

                // send the childID on the write-descriptor.
                childID = 1;
                write(fd[1], &childID, sizeof(childID));
                printf("Parent(%d) send childID: %d\n", getpid(), childID);

                // close the write descriptor
                close(fd[1]);
        } else {
                // child: reading only, so close the write-descriptor
                close(fd[1]);

                // now read the data (will block until it succeeds)
                read(fd[0], &childID, sizeof(childID));
```

```
                printf("Child(%d) received childID: %d\n", getpid(), childID);


                // close the read-descriptor
                close(fd[0]);
        }
        return 0;
}
```

Output:

aditi@aditi-Lenovo-ideapad-330S-14IKB-U:~/Desktop/UOS/13$ gedit A13_a.c

^C

aditi@aditi-Lenovo-ideapad-330S-14IKB-U:~/Desktop/UOS/13$ gcc A13_a.c

aaditi@aditi-Lenovo-ideapad-330S-14IKB-U:~/Desktop/UOS/13$ ./a.out

Parent(23699) send childID: 1

Child(23700) received childID: 1

aditi@aditi-Lenovo-ideapad-330S-14IKB-U:~/Desktop/UOS/13$ ^C

aditi@aditi-Lenovo-ideapad-330S-14IKB-U:~/Desktop/UOS/13$

Conclusion:

The concept of pipe has been learned.

References:

- https://bytefreaks.net/programming-2/c-programming-2/cc-pass-value-from-parent-to-child-after-fork-via-a-pipe
- https://notes.shichao.io/apue/ch15/