Unix pipes

Kjell Post
Institutionen för Datavetenskap
Tekniska Högskolan i Linköping

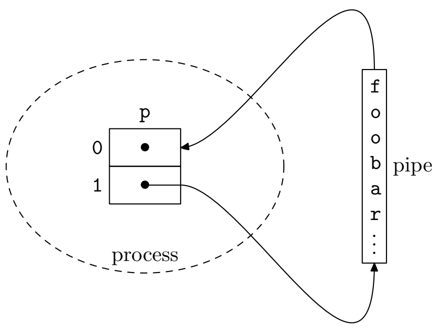
1987-09-16

Abstract

This memorandum describes Unix pipes and serves as a background explanation for the first lab assignment in Concurrent Programming. I wrote this in Troff as an undergrad, and recently revised it using LaTeX.

1 What is a pipe?

A system call to `pipe()` creates a communication channel, represented by two file descriptors. The two file descriptors will henceforth be called the *read end* and the *write end*. By writing characters to the write end, we place data in the pipe. Reading from the read end removes data from the pipe. It is common to place the two descriptors in a two-element array where the read- and write descriptors have index 0 and 1, respectively:



```
int p[2];
if (pipe(p)) {
   /* error */
}
```

2 What are open, close, write and read?

The system calls `open`, `close`, `write`, and `read` behave slightly different when the arguments are descriptors to pipes:

1. `write(`*descriptor*, *buffer*, *n*`)`

   `write` attemps to write *n* characters from the buffer into the pipe. If the pipe is full, `write` is *blocked* until characters have been removed by a `read` operation. The pipe's capacity is typically 64 kB. End-of-file is indicated by closing the write end. If successful, `write` returns the number of characters written.

2. `read(`*descriptor*, *buffer*, *n*`)`

   `read` tries to read *n* characters from the pipe, in the same order as they were written, and place them in the buffer. When characters are read, they are removed from the pipe. If the pipe is empty, the `read` operation is blocked until a character is available, unless the write descriptor has been closed in which case `read` returns 0 (end-of-file). If successful, `read` returns the number of characters read.

3. `open(`*descriptor*`)`

   `open` is not used with pipes. Use the `pipe` function instead.

4. `close(`*descriptor*`)`

   `close` releases a pipe descriptor. If the write end is closed, it will signal an end-of-file to the read end. If the read end is closed, any successive `write` operation will fail.

   A *deadlock* can occur if the program is incorrect: if a process does not close its write end, a naive reader can wait forever; in a similar vein, a passive reader can lead to the pipe being filled up and the writer being blocked.

3 What is fork?

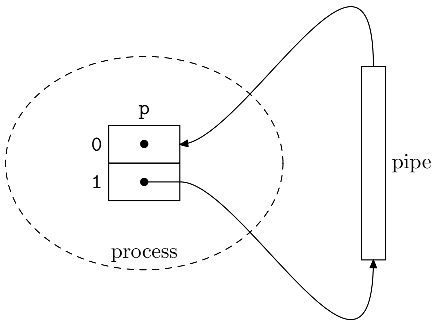
A new process is created by calling `fork()`. With a few exceptions, the new process (the child) is an exact copy of the process that called `fork()` (the parent). The value returned by `fork()` is different in the child and parent processes: the child receives 0 while the parent receives the process id of the child. This is enough information to let the child and parent do different things after the `fork()`.

Other differences are the process id (the child receives a new unique process id), and the execution time (the child's total execution is reset). However, any file descriptors created before `fork()` are shared.

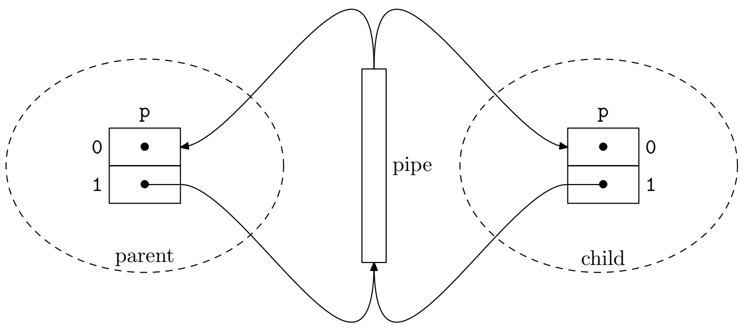
4 Communicating over a pipe

Given two processes, how can we connect them so that one process reads what the other one writes? The simple answer is: we can't. The pipe must be created before the processes fork. After the fork, the child inherits the pipe descriptors.

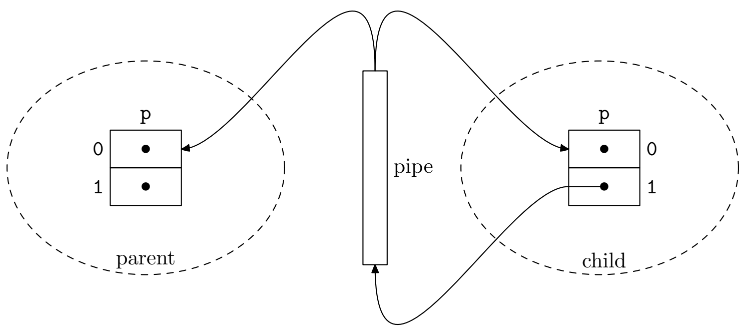
Let's look at an example where the child is the writer and the parent is the reader.



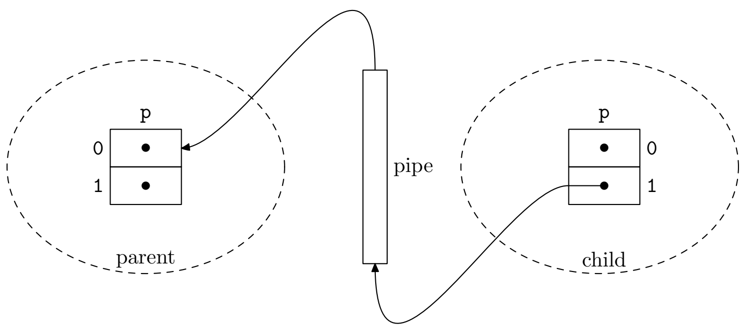
After calling `fork`:



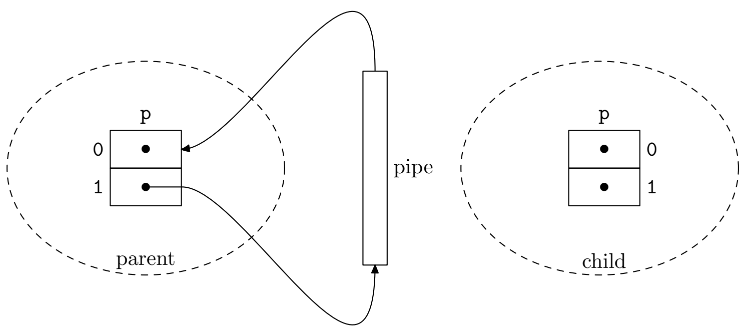
Because the parent is only reading and not writing, it starts by closing its write end. This will not affect the child's descriptor.



Similarily, the child has no use for its read end which can also be closed.



Note that the parent has a very good reason to close his write end: if the child terminates, its pipe descriptors will close:



But there's still a reference to the write end, namely the one that the parent has. When the parent calls `read`, it will be blocked forever because it is waiting for a writer, which happens to be the parent itself!

5 A solution in C

The following C program, written by Mikael Pettersson (IDA) shows a solution to the first lab assignment. With the previous explanations you will hardly need to understand C in order to grasp the concepts.

```
#include <unistd.h>
#include <stdlib.h>

#define STDIN 0
#define STDOUT 1
#define STDERR 2
#define READ 0
#define WRITE 1
#define CHILD 0
#define NEWLINE '\n'
```

```
#define OK 0
#define ERROR 1

int main() {
  int p1[2], p2[2];             /* two pipes, each with two ends */
  char buf[1];

  if (pipe(p1) || pipe(p2)) {   /* create two pipes */
    write(STDERR, "Couldn't pipe\n", 14);
    exit(ERROR);
  }

  if (fork() == CHILD) {        /* this becomes process 1 */
    close(STDOUT);   close(p1[READ]);
    close(p2[READ]); close(p2[WRITE]);
    while (read(STDIN, buf, 1)) {
      if (buf[0] == NEWLINE)
        buf[0] = ' ';
      write(p1[WRITE], buf, 1);
    }
    close(STDIN);    close(p1[WRITE]);
    exit(OK);
  } else if (fork() == CHILD) { /* this becomes process 2 */
    close(STDIN);     close(STDOUT);
    close(p1[WRITE]); close(p2[READ]);
    while (read(p1[READ], buf, 1)) {
      if (buf[0] != ' ')
        write(p2[WRITE], buf, 1);
    }
    close(p1[READ]);  close(p2[WRITE]);
    exit(OK);
  } else {                      /* parent continues here and becomes process 3 */
    close(STDIN);     close(p1[READ]);
    close(p1[WRITE]); close(p2[WRITE]);
    int n;
    while (read(p2[READ], buf, 1)) {
      if (('a' <= buf[0]) && (buf[0] <= 'z'))
        buf[0] -= 32;
      write(STDOUT, buf, 1);
    }
    close(p2[READ]);  close(STDOUT);
    exit(OK);
  }
}

$ ./lab1
fie foo fum
FIEFOOFUM^D
$
```