pipex tutorial — 42 project

Pipex is a project that reproduces the behaviour of the shell pipe `|` command in C, using the `pipe()` , `fork()` , `dup2()` and `execve()` functions. The program will launch as `./pipex infile cmd1 cmd2 outfile` and behave exactly as this line in the shell `< infile cmd1 | cmd2 > outfile` .

This article is structured as follow:
1 — Background theory — `pipe()` , `fork()` , `dup2()` and `execve()`
2 — How to do pipex with two child processes (and why you should)
3 — The `access()` function

4 — Frequently encountered problems

Photo by Gordon Johnson from Pixabay

1 — Background theory — pipe(), fork(), dup2() and execve()

Visually, this is the order in which these functions will be used:

```
# this is only a simplified sketch, refer to section 2 later on
# ./pipex infile cmd1 cmd2 outfile


pipe()
 |
 |-- fork()
      |
      |-- child // cmd1
      :     |--dup2()
      :     |--close end[0]
      :     |--execve(cmd1)
      :
      |-- parent // cmd2
            |--dup2()
            |--close end[1]
            |--execve(cmd2)
```

The general idea: we read from *infile*, execute cmd1 with infile as input, send the output to cmd2, which will write to *outfile*.
We need `pipe()` to send the output of the first `execve()` as input to the second `execve()`; we need `fork()` in order to run two processes (i.e. two commands) in one single program; we need `dup2()` to swap our files

with stdin and stdout. And of course a bit of parsing.
Visually,

```
// each cmd needs a stdin (input) and returns an output (to stdout)

    infile                                             outfile
as stdin for cmd1                                 as stdout for cmd2
       |                        PIPE                        ↑
       |           |---------------------------|            |
       ↓             |                     |             |
      cmd1   -->    end[1]       ↔       end[0]   -->     cmd2
                    |                     |
           cmd1   |---------------------------|  end[0]
          output                             reads end[1]
        is written                          and sends cmd1
         to end[1]                           output to cmd2
       (end[1] becomes                            (end[0] becomes
        cmd1 stdout)                              cmd2 stdin)
```

. . .

`pipe()` takes an array of two int such as `int end[2]`, and links them together. In a pipe, what is done in end[0] is visible to end[1], and vice versa. Plus, `pipe()` assigns an *fd* to each end.
Fd are file descriptors, and since files can be read and written to, by getting an fd each, the two ends can communicate: end[1] will write to the its own fd, and end[0] will be able to see and read end[1]'s fd and write to its own.

```
void    pipex(int f1, int f2)
{
    int end[2];


    pipe(end);
}
```

So now that we have a pipe, we need one process to populate each end.
`fork()` will split our process in two sub-processes: it returns 0 for the child process, a non-zero number for the parent process, or a -1 in case of error.
Also: `fork()` splits the process in two *parallel*, *simultaneous* processes, that happen at the *same* time. This will be important for section 2.

```
void    pipex(int f1, int f2)
{
    int   end[2];
    pid_t parent;

    pipe(end);
    parent = fork();
    if (parent < 0)
         return (perror("Fork: "));
    if (!parent) // if fork() returns 0, we are in the child process
        child_process(f1, cmd1);
    else
        parent_process(f2, cmd2);
}
```

Once *inside* the pipe, everything we do will go to one of its ends, one end will write and the other will read (small note on this, see section 4).
From now on we will say that end[1] is the child process, and end[0] the parent process, and that the child writes, while the parent reads. And since for something to be read, it must be written first, so cmd1 will be executed by the child, and cmd2 by the parent.

Earlier on, we said that pipe creates fds.
Because we run pipex like this `./pipex infile cmd1 cmd2 outfile` , we actually already have some fds (the infile and the outfile), and we want them to be the stdin and stdout of the pipe.
On linux, you can check your fds currently open with the command `ls -la /proc/$$/fd` (0, 1 and 2 are by default assigned to stdin, stdout and stderr). Our fd table right now looks like this:

```
            -----------------
   0        |    stdin     |
            -----------------
   1        |    stdout    |
            -----------------
   2        |    stderr    |
            -----------------
   3        |    infile    |  // open()
            -----------------
   4        |    outfile   |  // open()
            -----------------
   5        |    end[0]    |
            -----------------
   6        |    end[1]    |
            -----------------
```

For the *child* process, we want *infile* to be our stdin (we want it as input), and end[1] to be our stdout (we want to write to end[1] the

output of cmd1, so that the parent will be able to read it).
In the *parent* process, we want end[0] to be our stdin (end[0] at this point has already read from end[1] the output of cmd1), and *outfile* to be our stdout (we want to write to it the output of cmd2).

. . .

`dup2()` can swap our fds to stdin/stdout.
From the MAN, `int dup2(int fd1, int fd2)` : it will close fd2 and duplicate the value of fd2 to fd1, or else said, it will redirect fd1 to fd2.

For example, in the child process, in pseudo code:

```
# child_process(f1, cmd1);


// add protection if dup2() < 0
// dup2 close stdin, f1 becomes the new stdin
dup2(f1, STDIN_FILENO); // we want f1 to be execve() input
dup2(end[1], STDOUT_FILENO); // we want end[1] to be execve() stdout
close(end[0]) # --> always close the end of the pipe you don't use,
                    as long as the pipe is open, the other end will
                    be waiting for some kind of input and will not
                    be able to finish its process
close(f1)
// execve function for each possible path (see below)
exit(EXIT_FAILURE);
```

Our fd tables would now look like this:

```
          -----------------
  0       |    infile *  |  (x stdin closed)
          -----------------
  1       |    end[1] *  |  (x stdout closed)
          -----------------
  2       |    stderr    |
          -----------------
  3       |    infile    |
          -----------------
  4       |    outfile   |
          -----------------
  5       |    end[0]    |
          -----------------
  6       |    end[1]    |
          -----------------          *duplicated
```

Parent process in pseudo code code will be similar, but it needs a `waitpid()` at the very beginning to wait for the child to finish her process.

```
# parent_process(f2, cmd2);

int status;

waitpid(-1, &status, 0);
dup2(f2, ...); // f2 is the stdout
dup2(end[0], ...); // end[0] is the stdin
close(end[1])
close(f2);
// execve function for each possible path (see below)
exit(EXIT_FAILURE);
```

. . .

With the right stdin and stdout, we can execute the command with `execve()` , which needs a bit of parsing.

> *From the MAN,*
> ***int execve(const char *path, char *const argv[], char *envp[]);***

```
int execve(const char *path, char *const argv[], char *envp[]);

# path: the path to our command
        type `which ls` and `which wc` in your terminal
        you'll see the exact path to the commands' binaries

# argv[]: the args the command needs, for ex. `ls -la`
          you can use your ft_split to obtain a char **
          like this { "ls", "-la", NULL }
          it must be null terminated

# envp: the environmental variable
        you can simply retrieve it in your main (see below)
        and pass it onto execve, no need to do anything here
        in envp you'll see a line PATH which contains all possible
        paths to the commands' binaries

int main(int ac, char **ag, char **envp)
{
     int f1;
     int f2;

     f1 = open(ag[1], O_RDONLY);
     f2 = open(ag[4], O_CREAT | O_RDWR | O_TRUNC, 0644);
     if (f1 < 0 || f2 < 0)
          return (-1);
     pipex(f1, f2, ag, envp);
     return (0);
}
```

To see what is inside `envp`, type `env` in your terminal.
You’ll see a line `PATH` , those are all the possible paths to the command binaries.
You’ll need to split: you can use `:` as a delimiter, and retrieve them (small note on this in section 4).
Your execve function will have to try every possible path to the cmd until it finds the good one.
To see the path to the command `ls`, for ex., you can type `which ls` in your terminal. If the command does not exist, execve will do nothing and return -1; else, it will execute the cmd, delete all ongoing processes (including variables, so no need to worry about free) and exit (more on this, see section 3 below). In pseudo code,

```
// parsing (somewhere in your code)

char *PATH_from_envp;
char **mypaths;
char **mycmdargs;

// retrieve the line PATH from envp
PATH_from_envp = ft_substr(envp ....);
mypaths = ft_split(PATH_from_envp, ":"); // see section 4 for a
                                       small note[0]
mycmdargs = ft_split(ag[2], " ");

// in your child or parent process

int  i;
char *cmd;

i = -1;
while (mypaths[++i])
{
    cmd = ft_join(mypaths[i], ag[2]); // protect your ft_join
    execve(cmd, mycmdargs, envp); // if execve succeeds, it exits
    // perror("Error"); <- add perror to debug
    free(cmd) // if execve fails, we free and we try a new path
}
return (EXIT_FAILURE);
```

2 — How to do pipex with two child processes (and why you should)

Like this, with one child and one parent process, your pipex will work, but it is not super protected. Since the two processes happen at the same time, if one segfaults, the error will be sent to the other. If instead we divide the workload on different children, the parent can just wait for the job to be done and supervise the children’s *status*.

We will thus have to fork twice, and assign child1 to execute cmd1, and child2 to execute cmd2. The parent will wait at the end. In pseudo code,

```
void    pipex(int f1, int f2, char *cmd1, char *cmd 2)
{
    int   end[2];
    int   status;
    pid_t child1;
    pid_t child2;


    pipe(end);
    child1 = fork();
    if (child1 < 0)
         return (perror("Fork: "));
    if (child1 == 0)
        child_one(f1, cmd1);
    child2 = fork();
    if (child2 < 0)
         return (perror("Fork: "));
    if (child2 == 0)
        child_two(f2, cmd2);
    close(end[0]);         // this is the parent
    close(end[1]);         // doing nothing
    waitpid(child1, &status, 0);  // supervising the children
    waitpid(child2, &status, 0);  // while they finish their tasks
}
```

3 — The `access()` function

If you run `< infile pikachu | ls > oufile` in the shell, you'll get `-bash: pikachu: command not found`.
If you now run `./pipex infile pikachu ls outfile`, your program will do nothing and exit without error messages. `execve()` will execute nothing if the command is not found. You need to check if it exists before its execution.
To do this, check the `access()` function. With the proper mode, you can use it to verify whether the command exists and is *executable*.

4 — Frequently encountered problems

[0] As for splitting your envp path, print out the result of your split and have a look. Add a `/` at the end for the path to work correctly.

[1] If you launch your program and it gets stuck without executing anything, you most probably have not closed the ends of your pipe correctly. Until one end is open, the other will be waiting for input and its process will not finish.

[2] Place `perror("Error")` here and there in your code, especially right after `fork()` or `execve()` , to see what is going on in the pipe. Inside the pipe, everything we do will go to one of its ends. If we use printf, it won't show in the terminal or it will print to your outfile (because we swapped the stdout); `perror("Error")` will work because it prints to stderr.

[3] Handle file rights when you `open()` them. Return error if the file cannot be opened, read or written. Check how the shell treats infile and outfile when they do not exist, are not readable, writable etc. (`chmod` is your best friend).

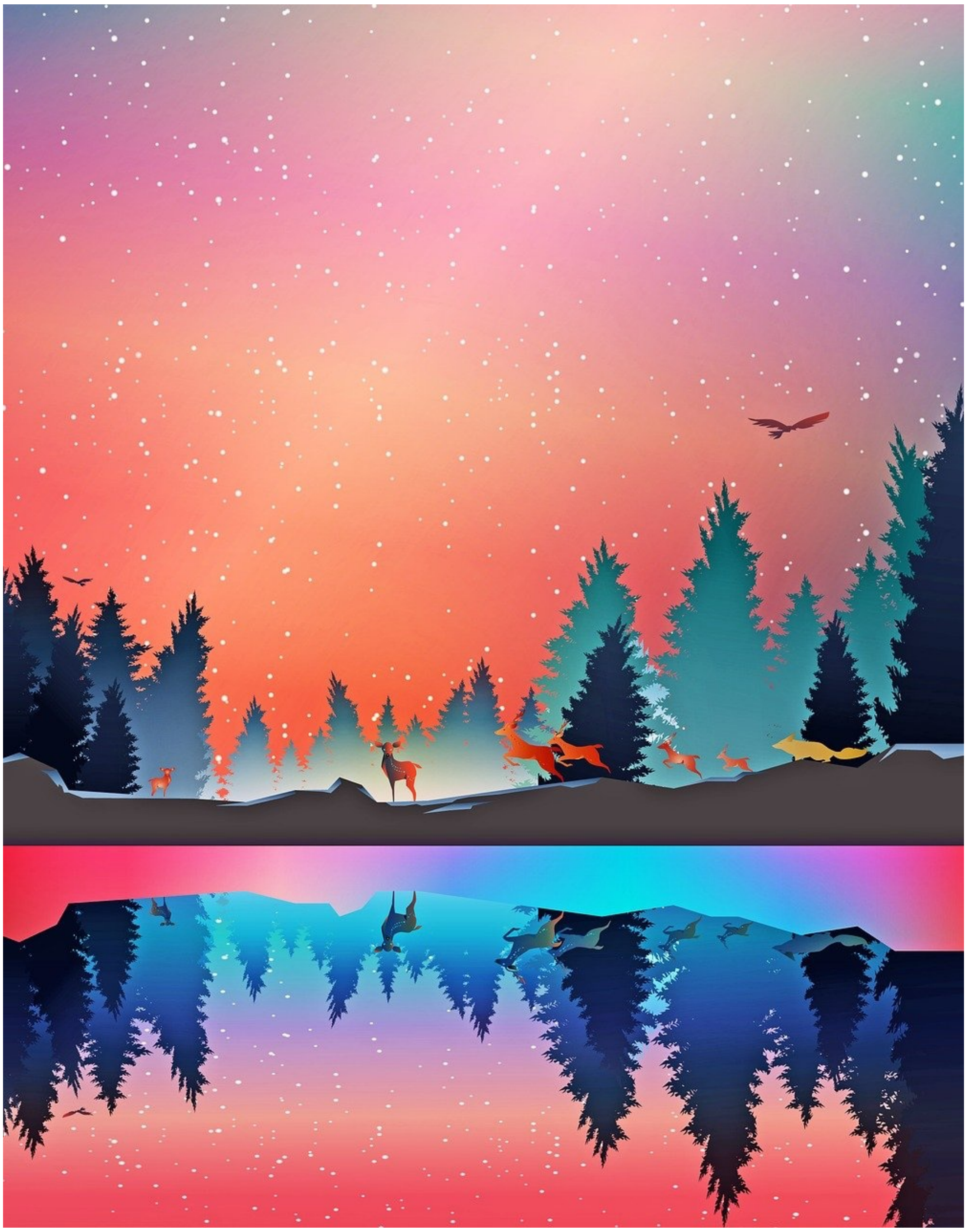
Photo by Angela Rose from Pixabay