UMEÅ UNIVERSITY **September 20, 2021**
Department of Computing Science
Assignment 1 report

5DV088 - Assignment 1, v1.0

C programming and Unix, Autumn 2021, 7.5 Credits

Pipeline implementation and algorithm

Name	Elias Olofsson
user@cs.umu.se	`tfy17eon@cs.umu.se`

Teacher
Mikael Rännar

Graders
Jonas Ernerstedt, Elias Häreskog, Oscar Kamf

Contents

1 Introduction

In this lab we take a look at how to write a program which can execute an arbitrary *pipeline* of programs in the programming language C. Since many pre-existing programs are constructed to read from and write to some of the standard I/O streams *standard input* (`stdin`), *standard output* (`stdout`) and *standard error* (`stderr`), it is convenient to take multiple of these programs and redirect the in- and output streams such that the programs are reading and writing to each other in a sequence. This allows us to sequentially process data through a multitude of otherwise independent programs, by chaining them together in a pipeline. Most shells already have built-in support for executing pipelines, e.g. in `bash` with the "pipe-symbol" `|`, but here we will perform an exercise in creating our own C-program to accomplish the same task.

2 System description

The final C-program `mexec.c` I have written can broadly be described by the pseudo-code seen in the figure below. I have not supplied a diagram of function calls since I have written the whole program in the main function, without implementing any auxiliary functions. The main reason for doing this is that the program as a whole is rather short, and I did not see any obvious way of segmenting the program into component functions which would, in a substantial way, improve the readability of the code or remove any unnecessarily duplicated lines of code. Since I was also a bit short on time, I thus actively chose to not retroactively implement any auxiliary functions after already having written the working program in the main function. I realize this is not the best practice, but at this time I do not have much of a choice.

However, the program itself starts by first selecting whether to read from `stdin` or from a file supplemented as an argument at time of program execution. The number of arguments (zero or one) determines which case it is, and the rest of the program functions identically regardless if we read from a file or from `stdin`. We then read one line at a time from the specified input stream until we reach *end-of-file* (`EOF`). Each line gets parsed by "splitting" up the full string into separate strings or *tokens*, delimited by whitespace characters. Each line we parse correspond to an individual program command and associated program arguments. When each line have been read, parsed and appropriately saved to an "2d array" we have created to contain the commands, we continue to the next step of creating pipes.

mexec.c: main()

```
// Read from stdin, or open and read from file.
input = stdin or file

// Read one line at a time into a buffer and parse it.
n = 0
while (buf = read_line(input) and input not EOF) do
    // Split buffer into separate strings, delimited by whitespace
    i = 0
    while not end of buffer, do
        token = split(buf)
        args[i++] = token // Resize array & append
    program_commands[n++][] = args // Resize 2D array & append

// Open all pipes
for i in (n-1) program commands do
    pipeID[i] = pipe() // Append to array

// Create children processes
for child_id in (n) program commands, do
    pid = fork()
    if pid is child, do
        if child_id is first, do
            dup2(pipeID[child_id][WRITE] to stdout)
        else if child_id is last, do
            dup2(pipeID[child_id-1][READ] to stdin)
        else do
            dup2(pipeID[child_id-1][READ] to stdin)
            dup2(pipeID[child_id][WRITE] to stdout)
        break loop

// Close all pipes
for i in (n-1) program commands, do
    close(pipeID[i])

// If child, execute parsed program command
if (pid is child) do
    exec(program_commands[child_id])

// Let parent wait on its children
for (n) program commands, do
    wait()
```

Since each line we have parsed corresponds to one program execution, and we want the pipes to funnel data between each other in a linear and sequential manner, we need to create $n-1$ pipes for the n program commands.

When the $n-1$ pipes are initialized, we then need to create n new processes in which we can execute the program commands. For each of the n children processes required, we perform a call to `fork` which duplicates the current process. Immediately after the fork, we use an `if`-statement to differentiate between the child and parent processes, such that we can control the behaviour of them both independently. Since we want the parent process to be the parent of all n child processes, we need to introduce a `break`-statement to let the child exit out of the `fork`-loop, while the parent continues spawning more children processes.

However, at this time before the `break`-statement, it is an opportune moment to let the child duplicate the file descriptors associated with the pipes we created earlier. This is the important step which allows us to redirect the `stdin` and `stdout` streams of the individual programs and enables piping of data between processes. We use the for-loop index variable to keep track of the relative order between the children processes, which determines which file descriptors we duplicate for each process. In the first child, which is supposed to run the first program command, we leave the `stdin` stream untouched and only duplicate the write-end of the first pipe to replace `stdout`, closing the original standard output stream of the process. Conversely in the last child, which will execute the last program in the pipeline, we leave `stdout` as-is and duplicate the read-end of the last pipe to `stdin`. In the intermediate children, we perform two duplications to replace the file descriptors of both `stdin` and `stdout` by the read-end of the previous pipe and the write-end of the next pipe.

If this concert of file descriptor duplication is performed correctly, the first child should now be able to read from `stdin` and write data to the next child process in line. Each next children should be able to read data from another process, and send data further to the next process after it. The last process has the option to output data to the normal `stdout`, and the pipeline is done.

However, before any program is actually executed and data is transferred, it is a good practice to close all file descriptors which are not used, and should not be used, in each individual process. Since we have duplicated all pipe-ends to the old positions of the `stdin` or `stdout` streams in each process, we can safely close all file descriptors which we opened when we originally initialized the pipes. Because these descriptors were already duplicated, the pipes of the pipeline will still be in place and in order, but now there is no risk of processes writing or reading data from the wrong pipe-ends and

creating problems.

Finally, it is time to actually execute the program commands we originally parsed at the beginning. We reuse the identifier from before to tell the relative order of the children, and let each children execute its corresponding command with associated arguments. At this point, my program is completely replaced by the new programs in the children processes, and thus none of the children will continue further down in the program, if no exception or error occurs. All that is left to do is to let the parent process wait on all of the children to finish their programs. Once all children have exited, the parent process exits as well.

This entire process of creating a pipeline can be shown in fig.(1,2), where I have made an attempt at visually describing the full process. I wanted to create the equivalent picture in `tikz` instead, but there was no time to finish such an undertaking.

3 Algorithm for creating pipes

The generic algorithm to create a pipe where information can be transferred between two different processes is to first initialize a pipe by calling the function `pipe()`, and then spawn a child process by calling the function `fork()`. Next, each process need to take care to close the appropriate pipes, such that there's not multiple processes reading from or writing to the same end of a pipe. Optionally, one can duplicate the file descriptors of the pipe such that they replace e.g. `stdin` or `stdout`, as we have done when creating the pipeline as described above in the previous section.

4 Discussion and reflection

This was quite a fun, but at times very frustrating exercise. One small detail cost me a lot of time troubleshooting potential mistakes and errors, namely I had simply used `fclose(input)` naively directly after the input parsing. This is correct if you are reading from a file, but since I let the file pointer `input` be `stdin` if no arguments was given to the program, I would then accidentally close `stdin` of the parent, which is then reflected in all of the children processes. I could not for the longest time figure out why I was getting errors and complaints of bad file descriptors, and only when I did not supply a file to the program. Anyhow, I really enjoyed this lab overall, and it was quite satisfying to finally get the program in order and correctly working.

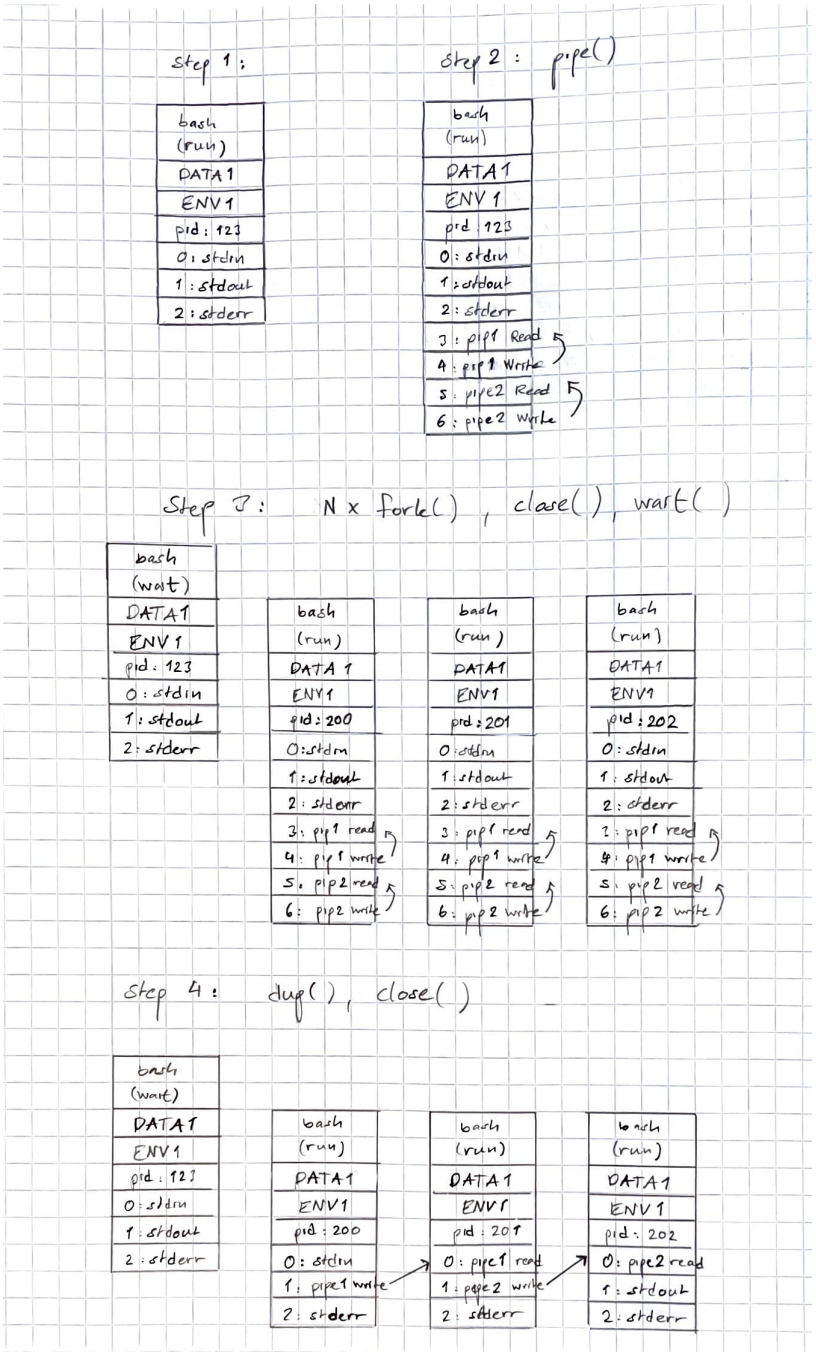

Figure 1 – The method of setting up a pipeline, part 1.

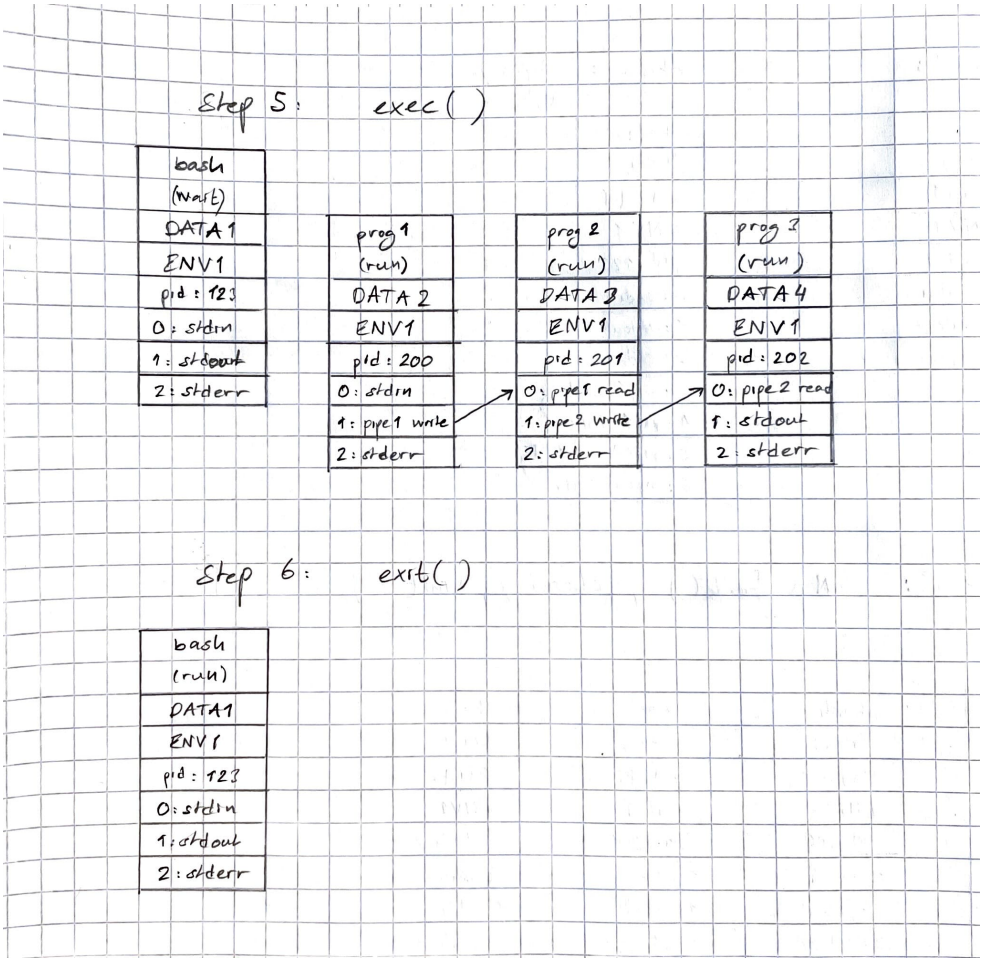

Figure 2 – The method of setting up a pipeline, part 2.