UMEÅ UNIVERSITY **October 10, 2021**
Department of Computing Science
Assignment 1 report

5DV088 - Assignment 1, v2.0

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 I have written is contained in `mexec.c`. Additionally, I implemented a dynamic, directed list data structure as a supplement for the main program, which is contained in `list.c` and its associated header-file `list.h`. These dependencies are reflected in the included `Makefile`, such that compilation of the main program `mexec` is done correctly.

The list data structure is inspired by the implementations of directed and undirected linked list data structures written by Niclas Börlin, Adam Dahlgren Lindström and Lars Karlsson, of the Department of Computing Science at Umeå University. Originally, these implementations were written for the "Data-structures and Algorithms" courses at Umeå University. Modifications I have made to the original code include; a few extra methods added to the general data structure and error checking which was lacking for the pre-existing methods.

The hierarchy of function calls performed in the program is described in Fig.(1) below, where a tree graph lays out the relative order and dependencies of each implemented function, both in the main program `mexec.c` and the list data-structure `list.c`.

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 go on to parsing the input stream by calling the function

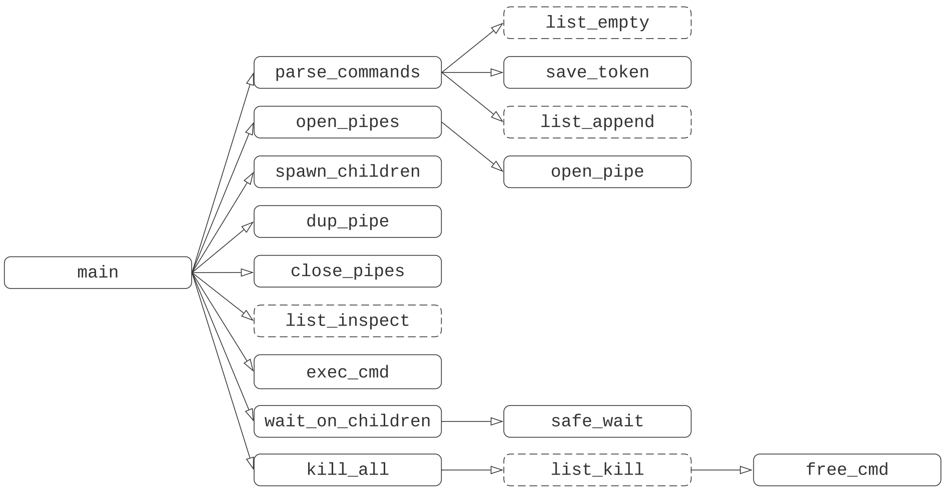

Figure 1 – Graph of function calls for the program `mexec.c`. An arrow from A to B implies that function A calls function B. The functions with a dashed outline are contained in the external data-structure `list.c`.

`parse_commands`, which is supposed to contain the program commands we want to execute in the pipeline. Each line should have its own individual program command with associated options and arguments. Thus, we read one line at a time from the input stream until we reach *end-of-file* (`EOF`), and parse each line by "splitting" the full string into separate strings or *tokens*, delimited by whitespace characters. We save each token string to an array of strings using the function `save_token`, which we then append to a list containing the program commands. Each element in the list represents one program command with associated arguments, which means that the length of the list corresponds to the number of children processes we need to create later. Also, since we want the pipes to funnel data between each of the children in a linear and sequential manner, we need to create $n-1$ pipes for the n program commands.

Next, we call function `open_pipes` to initialize the $n-1$ pipes we need, after which we call function `spawn_children` to create the n new processes in which we can execute the program commands. New processes are created by repeated calls to the standard C library function `fork`, which performs a system call that duplicates the calling process. Thus we let the parent process generate the n children processes required. Is it important to let the pipe-creation occur before creating new children processes, otherwise the pipe file descriptors generated will not be accessible in the children processes, and consequently data can not be funneled between the processes using the pipes. Additionally, we also set a sort of unique identifier for each of the processes, a single integer `child_no`, which helps us differentiate and keep the relative order between each of the children processes and the parent process.

After we have successfully generated all the children processes needed, we need to let each child appropriately 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 call the function `dup_pipe` and use the previously created integer identifier `child_no` 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 we want to utilize to the old positions of the `stdin` or `stdout` streams in each process, we can safely close all pipes in the file descriptor array we originally generated at initialization of the pipes, by calling the function `close_pipes` on the array. The benefit now is that we have eliminated the risk of any process writing or reading data from the wrong pipe-ends and potentially creating problems.

Finally, it is time to actually execute the program commands we originally parsed at the beginning. We reuse the identifier `child_no` from before to tell the relative order of the children, and fetch the corresponding program command by calling `list_inspect` on the list containing all program commands. Then we let each child execute its corresponding command by calling the function `exec_cmd`. At this point, the program `mexec` is completely replaced by the new program instances in all of the children processes, and thus none of the children will continue executing further down into `mexec`, unless any exception or error occurs. All that is left to do is to let the parent process wait on all of the children processes to finish their programs, which is done by letting the parent call the function `wait_on_children`. Once all children have exited, the parent process returns all dynamically allocated memory via a call to `kill_all`, and exits as well.

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 pipe-ends, such that multiple processes does not read from or write 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.

This entire process of creating a pipeline can be shown in fig.(2,3), 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.

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.

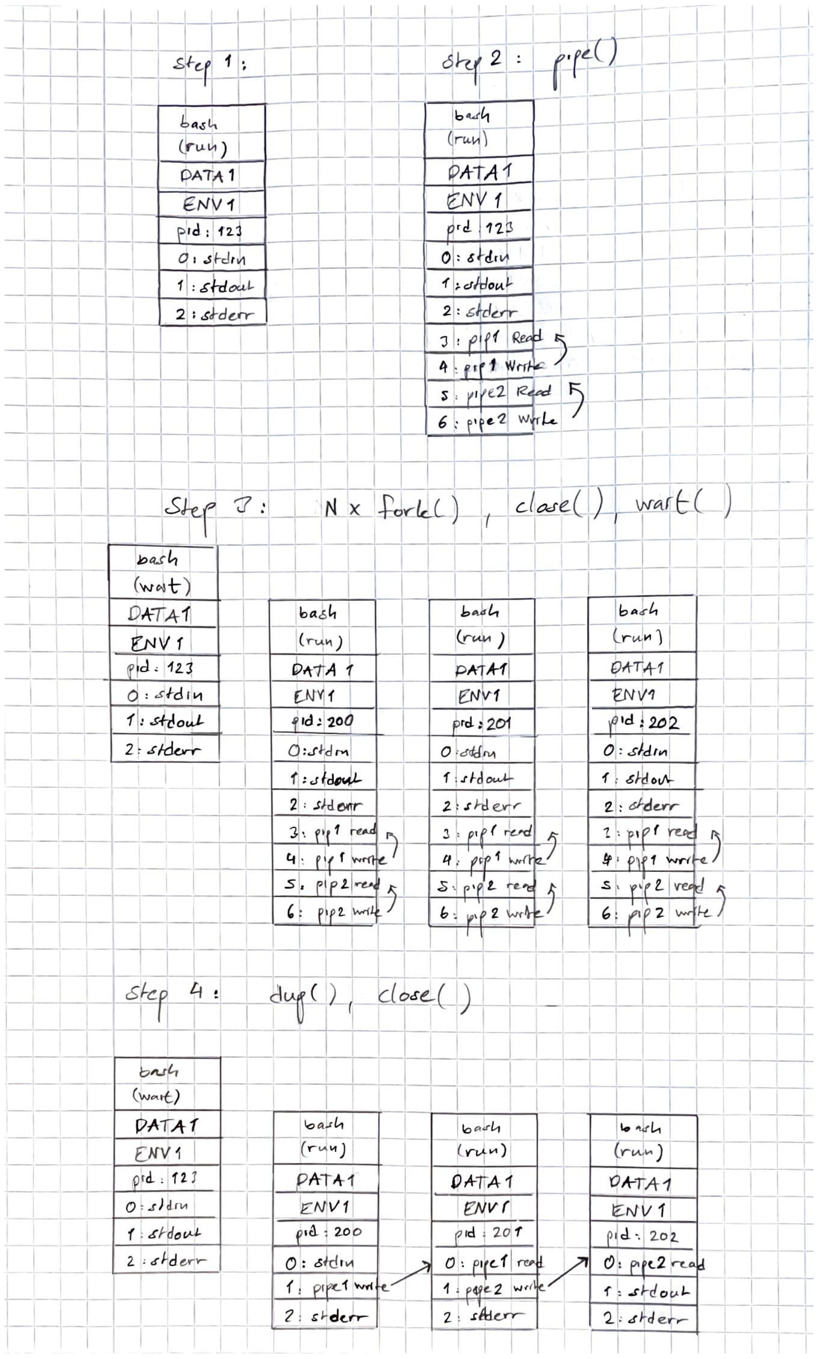

Figure 2 – The method of setting up a pipeline, part 1.

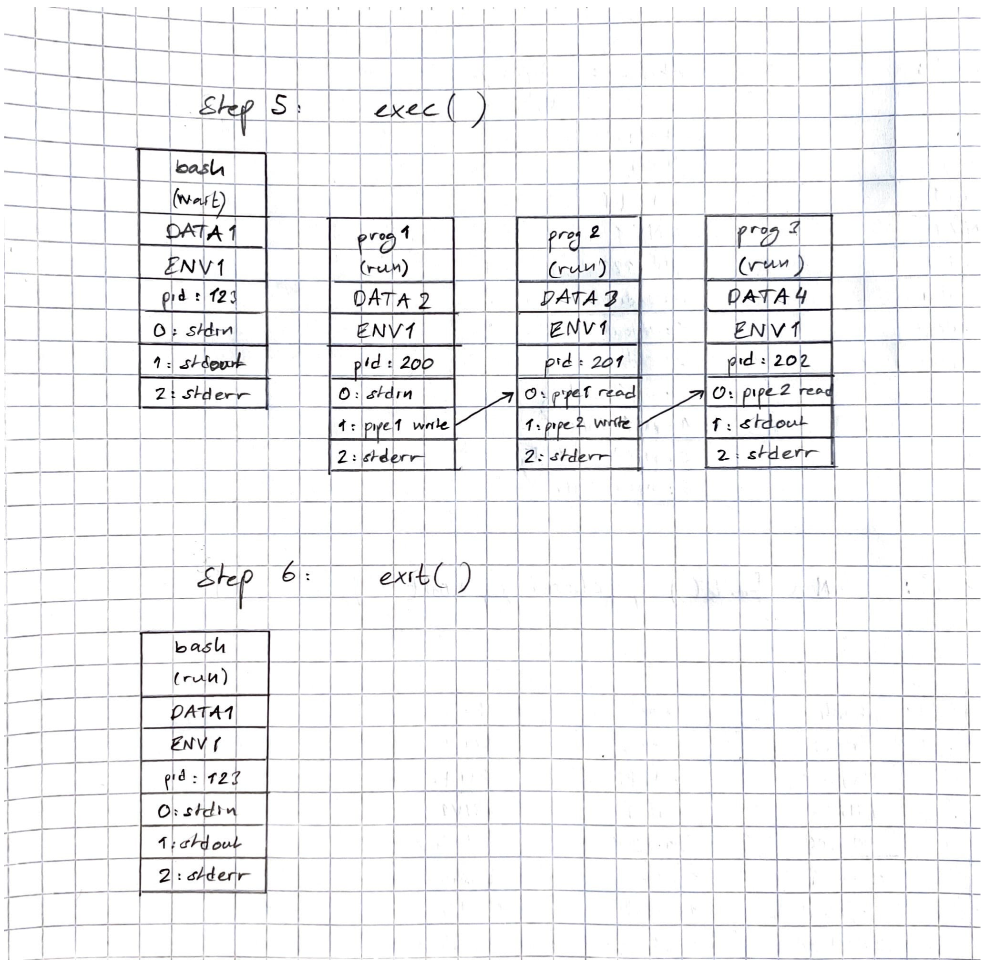

Figure 3 – The method of setting up a pipeline, part 2.