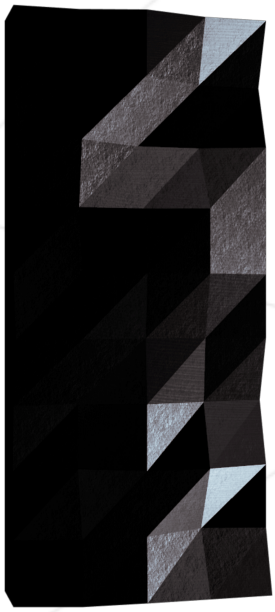

Get Next Line

Reading a line on a fd is way too tedious

Summary: The aim of this project is to make you code a function that returns a line ending with a newline, read from a file descriptor.

Contents

Chapter I

Goals

This project will not only allow you to add a very convenient function to your collection, but it will also allow you to learn a highly interesting new concept in `C` programming: static variables.

Chapter II

Common Instructions

- Your project must be written in accordance with the Norm. If you have bonus files/functions, they are included in the norm check and you will receive a `0` if there is a norm error inside.
- Your functions should not quit unexpectedly (segmentation fault, bus error, double free, etc) apart from undefined behaviors. If this happens, your project will be considered non functional and will receive a `0` during the evaluation.
- All heap allocated memory space must be properly freed when necessary. No leaks will be tolerated.
- If the subject requires it, you must submit a `Makefile` which will compile your source files to the required output with the flags `-Wall`, `-Wextra` and `-Werror`, and your Makefile must not relink.
- Your `Makefile` must at least contain the rules `$(NAME)`, `all`, `clean`, `fclean` and `re`.
- To turn in bonuses to your project, you must include a rule `bonus` to your Makefile, which will add all the various headers, librairies or functions that are forbidden on the main part of the project. Bonuses must be in a different file `_bonus.{c/h}`. Mandatory and bonus part evaluation is done separately.
- If your project allows you to use your `libft`, you must copy its sources and its associated `Makefile` in a `libft` folder with its associated Makefile. Your project's `Makefile` must compile the library by using its `Makefile`, then compile the project.
- We encourage you to create test programs for your project even though this work **won't have to be submitted and won't be graded**. It will give you a chance to easily test your work and your peers' work. You will find those tests especially useful during your defence. Indeed, during defence, you are free to use your tests and/or the tests of the peer you are evaluating.
- Submit your work to your assigned git repository. Only the work in the git repository will be graded. If Deepthought is assigned to grade your work, it will be done after your peer-evaluations. If an error happens in any section of your work during Deepthought's grading, the evaluation will stop.

Chapter III

Mandatory part - Get_next_line

Function name	`get_next_line`
Prototype	`int get_next_line(int fd, char **line);`
Turn in files	`get_next_line.c, get_next_line_utils.c, get_next_line.h`
Parameters	`#1. file descriptor for reading` `#2. The value of what has been read`

Return value	`1 : A line has been read`
	`0 : EOF has been reached`
	`-1 : An error happened`

External functs.	`read, malloc, free`
Description	`Write a function which returns a line read from a file descriptor, without the newline.`

- Calling your function `get_next_line` in a loop will then allow you to read the text available on a file descriptor one line at a time until the EOF.
- Make sure that your function behaves well when it reads from a file and when it reads from the standard input.
- `libft` is not allowed for this project. You must add a `get_next_line_utils.c` file which will contain the functions that are needed for your get_next_line to work.
- Your program must compile with the flag `-D BUFFER_SIZE=xx`. which will be used as the buffer size for the read calls in your get_next_line. This value will be modified by your evaluators and by moulinette.
- Compilation will be done this way : `gcc -Wall -Wextra -Werror -D BUFFER_SIZE=32 get_next_line.c get_next_line_utils.c`
- Your `read` must use the BUFFER_SIZE defined during compilation to read from a file or from stdin.
- In the header file `get_next_line.h` you must have at least the prototype of the function `get_next_line`.

Does your function still work if the BUFFER_SIZE value is 9999? And if the BUFFER_SIZE value is 1? And 10000000? Do you know why?

You should try to read as little as possible each time get_next_line is called. If you encounter a newline, you have to return the current line. Don't read the whole file and then process each line.

Don't turn in your project without testing. There are many tests to run, cover your bases. Try to read from a file, from a redirection, from standard input. How does your program behave when you send a newline to the standard output? And CTRL-D?

- We consider that `get_next_line` has an undefined behavior if, between two calls, the same file descriptor switches to a different file before EOF has been reached on the first fd.
- `lseek` is not an allowed function. File reading must be done only once.
- Finally we consider that `get_next_line` has an undefined behavior when reading from a binary file. However, if you wish, you can make this behavior coherent.
- Global variables are forbidden.

A good start would be to know what a static variable is:
https://en.wikipedia.org/wiki/Static_variable

Chapter IV

Bonus part

The project `get_next_line` is straightforward and leaves very little room for bonuses, but I am sure that you have a lot of imagination. If you aced perfectly the mandatory part, then by all means complete this bonus part to go further. I repeat, no bonus will be taken into consideration if the mandatory part isn't perfect.
Turn-in all 3 initial files with `_bonus` for this part.

- To succeed `get_next_line` with a single static variable.
- To be able to manage multiple file descriptor with your `get_next_line`. For example, if the file descriptors 3, 4 and 5 are accessible for reading, then you can call `get_next_line` once on 3, once on 4, once again on 3 then once on 5 etc. without losing the reading thread on each of the descriptors.