Assignment 1: Command-line Memory
CSE 130: Principles of Computer Systems Design

Due: April 19, 2023 at 11:59 PM

Goal This project serves as a refresher for building software in 'C'. In particular, the project will involve a review of Linux system calls, buffering file I/O, memory management, and c-string parsing.

Assignment Details

For this assignment, you will write a program, `memory`, that provides a `get/set` memory abstraction for files in a Linux directory. Your program should take a command from `stdin` and carry out the command in the current working directory. Below, we provide examples of how `memory` should work, outline the commands formally, describe how to handle invalid commands, and provide notes about other functionality and limitations on `memory`.

Examples

Here, we illustrate how your server should operate with four examples. In each example, suppose that the user starts `memory` from a directory that includes the file `foo.txt`, with content "`Hello from foo`", and the file `bar.txt`, with content "`Hello from bar`".

1. Example 1. In this example, the user passes the string "`get foo.txt\n`" to `stdin`. `memory` outputs the contents of `foo.txt` ("`Hello from foo`") to `stdout` and then exit with a return code of 0.
2. Example 2. In this example, the user passes the string "`set baz.txt\nHello World!`" to `stdin`. `memory` should create a new file, named `baz.txt`, in the current directory, write "`Hello World!`" to `baz.txt`, write the message `OK` to `stdout`, and then exit with a return code of 0.
3. Example 3. In this example, the user passes the string "`set foo.txt\nHello World!`" to `stdin`. `memory` should change the contents of `foo` to be "`Hello World!`", write the message `OK` to `stdout`, and then exit with a return code of 0.
4. Example 4. In this example, a user passes the string "`invalid foo.txt`" to `stdin`. `memory` should output the message "`Invalid Command\n`" to `stderr`, and exit with a return code of 1.

Commands

Your program will receive a command from `stdin` as input. Each valid command with be either a `get` or `set` command.

- get. A valid get command is formatted as "`get <location>\n`" and indicates that `memory` should output the contents of the file named `location`. If there is a file named `location` in the current directory, then `memory` should write the contents of `location` to `stdout` and exit with a return code of 0.
- set. A valid set command is formatted as "`set <location>\n<contents>`" and indicates that `memory` should assign `contents` to a file named `location` in the current directory. If `location` is a valid filename (see Hints section for some tips on how to check for this), then `memory` should assign the content of `location` to be `content` (overwriting `location`'s current contents if applicable), and exit with return code 0.

Invalid Commands

If `memory` receives an invalid command, then it should output the text "`Invalid Command\n`" to `stderr` and exit with a return code of 1. An invalid command is one that meets any of the following:

- The command does not start with either `get` or `set` (these are case-sensitive).
- The command starts with `get` or `set`, but `stdin` was closed before the user provided a `location`.
- The command starts with `get` or `set`, includes a `location`, but the `location` is not a valid filename (see Hints section at the end of this document).
- The command is a `get` command, includes a valid `location`, but there is no file located at `location`.
- The command is a `get` command, includes a valid `location` that exists, but also includes additional input (e.g., "`get foo.txt\nExtraStuff`").

Notes

In addition to the functionality described above, you `memory` should meet the following functionality and limitations.

- If `memory` detects any other error (e.g. if it cannot write all of the requested content to a file for any reason), the program should produce the text `Operation Failed` to `stderr` and exit with a return code of 1.
- `memory` must be reasonably time efficient: it should buffer file input and output.
- `memory` must be reasonably space efficient: it should use at most 1MB of memory regardless of input.
- `memory` must not leak any memory (i.e., it should free all memory that it allocates).
- `memory` must not leak any file descriptors (i.e., it should close all files that it opens).
- `memory` should not crash (e.g., it should never `segfault`).
- `memory` must be written using the 'C' programming language (*not C++!*).
- `memory` cannot use the following functions from the 'C' `stdio.h` library: `fwrite`, `fread`, variants of `put` (i.e., `fptuc`, `putc`, `putc_unlocked`, `putchar`, `putchar_unlocked`, and `putw`), and `get` (i.e, `fgetc`, `getc`,`getc_unlocked`, `getchar`, `getchar_unlocked`, `getline`, and `getw`).
- `memory` cannot use functions, like `system(3)`, that execute external programs.

Rubric

We will use the following rubric for this assignment:

Category	Point Value
Makefile	10
Clang-Format	5
Files	5
Functionality	50
Total	70

Makefile Your repository includes a Makefile with the rules `all` and `memory`, which produce the `memory` binary, and the rule `clean`, which removes all `.o` and binary files. Additionally, your Makefile should use clang (i.e., it should set `CC=clang`), and should use the `-Wall`, `-Wextra`, `-Werror`, and `-pedantic` flags (i.e., it should set `CFLAGS=-Wall -Wextra -Werror -pedantic`).

Clang-Format All `.c` and `.h` files in your repository are formatted in accordance with the `.clang-format` file included in your repository.

Files The following files are included in your repository: `memory.c`, `Makefile`, and `README.md`. Your repository also should not include binary files nor any object files (i.e., `.o` files). To make it easier for you to maintain tests, we added an exception to the binary file rule: you *can* includes binary files in any directory whose name starts with the phrase `test`.

Functionality Your `memory` program performs the functionality described in the Assignment Details section.

How to submit

Submit a 40-character commit ID hash on the Canvas assignment to identify which commit you want us to grade. We will grade the last hash that you submit to the assignment on Canvas and will use the timestamp of your last upload to determine grace days. For example, if you post a commit hash 36 hours after the deadline, we will subtract 2 grace days from your total.

Resources

Here are some resources to help you:

Testing We provided you with two resources to test your own code:

1. An autograder, which is run each time you push code to GitLab, will show you the points that you will receive for your Makefile, Clang-Format, and Files.
2. A set of test scripts in the resources repository to check your functionality. You can use the tests to see if your functionality is correct by running them on your Ubuntu 22.04 virtual machine. We provided you with a subset of the tests that we will run, but, I bet you can figure the other ones out by adapting what we have given you :-)

Hints

- Check out the document "Setting up Ubuntu" on Canvas. It has some instructions for getting an Ubuntu 22.04 virtual machine up and running.
- To format a file, named `foo.c`, using the included `.clang-format`, execute the command `clang-format -i -style=file foo.c`. If you include the `.clang-format` file in the root directory of your repository (as it was when we created your repositories), you can instead execute `clang-format -i foo.c`.
- System calls that allow you to interact with standard input and output (e.g, `read` and `write`) will be helpful to achieve the functionality listed above. If you aren't sure how to use them, look up a function's manual page by typing `man <command>` in your terminal. Or, use Google!
- To check the return code of your program, run your program until completion and then type `echo $?` in your terminal.
- You can check your program for memory leaks using `valgrind --leak-check=full`.
- `memory` should allow a user to `get` or `set` data from any filename that is valid on a Linux filesystem. In Linux, the only restrictions on filenames are that they cannot be longer than `PATH_MAX` from the `"<limits.h>"` header and that they cannot include null characters (i.e., `\0`). To make your life easier, we're simplifying further: none of our tests that send input to your `memory` program and specify a valid method (i.e., either `get` or `set`) will include a filename with non-ASCII characters or spaces.
- This assignment requires that you perform some basic string parsing. There are many ways to parse a C-string; the most straightforward for this assignment is probably `strtok`, but you could also check out `regexec` or `sscanf`.