OPERATING SYSTEMS EXERCISE 1

Aufgabenstellung A – stegit

Eines kalten Herbstabends finden Sie am Bildschirm eine seltsame Botschaft, die Ihnen erklärt, dass Sie auserwählt wurden.

Beeindruckt beschließen Sie den Anweisungen darin sogleich Folge zu leisten – bevor Sie überhaupt noch wissen, worum es geht. Der Absender behauptet wichtige Informationen für Sie zu haben, benötigt aber schnellstens ein Tool, um Ihnen die geheime, möglicherweise existenzverändernde Botschaft zuschicken zu können. Sie erkennen sofort die Wichtigkeit der Situation und machen sich aufgeregt an die Implementierung des Programms `stegit`, um raschest die geheime Botschaft lesen zu können.

```
SYNOPSIS:
    stegit -f|-h [-o <filename>]

     -f                 find mode
     -h                 hide mode
    [-o <filename>]     output filename
```

Dieses Programm soll in zwei Modi ausgeführt werden können: *Verstecken* und *Finden*.

Im Modus *Verstecken* wird solange von *stdin* gelesen, bis *EOF* auftritt oder die Enter-Taste betätigt wird. Dann wird die eingelesene Botschaft in einem Text versteckt und auf *stdout* oder in die Ausgabedatei ausgegeben. Im Modus *Finden* soll wiederum von *stdin* so lange gelesen werden, bis *EOF* auftritt. Aus dem eingelesenen Text soll die geheime Botschaft ermittelt werden und entweder auf *stdout* angezeigt oder in die Ausgabedatei geschrieben werden. Falls eine Ausgabedatei angegeben wurde, muss dafür gesorgt werden, dass sie überschrieben wird, wenn sie schon vorhanden war; ansonsten soll sie neu angelegt werden.

Anleitung

Suchen Sie sich 28 beliebige Wörter und initialisieren Sie damit ein Zeichenketten-Array. Dieses soll sich fix im Programm befinden und nicht veränderbar sein. Mit Hilfe der Array-Indizes weisen Sie jedem Buchstaben im englischen Alphabet, sowie den Sonderzeichen *Leerzeichen* und *Punkt*, ein Wort zu. Damit sollte es kein Problem mehr sein, die eingelesene Botschaft in (unter Umständen) unsinnigen Text umzuwandeln. Sonderzeichen und nicht zugeordnete Zeichen werden ignoriert (geschluckt). Damit der resultierende End-Text „realistischer“ aussieht, bauen Sie zufällig (nach je 5 bis 15 Wörtern) Punkte ein – die Sie beim Einlesen im Find-Modus natürlich wiederum ignorieren. Die maximale Länge der geheimen Botschaft (Cleartext) können Sie auf 300 Zeichen begrenzen.

Beispieltabelle:

```
p -> "der"
s -> "Himmel"
c -> "ist"
h -> "heute"
t -> "klar"
. -> "la"
```

Die Geheimbotschaft:

```
pscht...
```

würde nach der Hide-Operation in folgenden Text resultieren (der Punkt ist Zufall):

```
der himmel ist heute klar. la la la
```

Testen

Starten Sie das Programm:

```
./stegit -h -o secret_message_within
```

und geben Sie folgenden Text ein (möglichst ohne dass Sie mitlesen!):

```
es muessen beide aufgabenstellungen a und b geloest werden
```

Drücken Sie die Enter-Taste und betrachten Sie die nun versteckte Botschaft mit:

```
cat secret_message_within
```

Versuchen Sie aus dem entstandenen Text mit:

```
./stegit -f < secret_message_within
```

die geheime Botschaft wiederherzustellen. Sie könnte Sie vor bösen Überraschungen bewahren.

Coding Rules and Guidelines

Your score depends upon the compliance of your submission to the presented guidelines and rules. Violations result in deductions of points. Hence, before submitting your solution, go through the following list and check if your program complies.

Rules

Compliance with these rules is essential to get any points for your submission. A violation of any of the following rules results in 0 points for your submission.

1. All source files of your program(s) must compile via

   ```
   $ gcc -std=c99 -pedantic -Wall -D_DEFAULT_SOURCE -D_BSD_SOURCE -D_SVID_SOURCE
                                  -D_POSIX_C_SOURCE=200809L -g -c filename.c
   ```

   without *errors* and your program(s) must link without *errors*. The compilation flags must be used in the Makefile. The feature test macros must not be bypassed (i.e., by undefining these macros or adding some in the C source code).

2. The functionality of the program(s) must conform to the assignment. The program(s) shall operate according to the specification/assignment given the test cases in the respective assignment.

General Guidelines

Violation of following guidelines leads to a deduction of points.

1. All source files of your program(s) must compile with

   ```
   $ gcc -std=c99 -pedantic -Wall -D_DEFAULT_SOURCE -D_BSD_SOURCE -D_SVID_SOURCE
                                  -D_POSIX_C_SOURCE=200809L -g -c filename.c
   ```

   without *warnings and info messages* and your program(s) must link without warnings.

2. There must be a Makefile implementing the targets: `all` to build the program(s) (i.e. generate executables) from the sources (this must be the first target in the Makefile); `clean` to delete all files that can be built from your sources with the Makefile.

3. The program shall operate according to the specification/assignment without major issues (e.g., segmentation fault, memory corruption).

4. Arguments have to be parsed according to UNIX conventions (we strongly encourage the use of `getopt(3)`). The program has to conform to the given synopsis/usage in the assignment. If the synopsis is violated (e.g., unspecified options or too many arguments), the program has to terminate with the usage message containing the program name and the correct calling syntax. Argument handling should also be implemented for programs without arguments.

5. Correct (=normal) termination, including a cleanup of resources.

6. Upon success the program has to terminate with exit code 0, in case of errors with an exit code greater than 0. We recommend to use the macros `EXIT_SUCCESS` and `EXIT_FAILURE` (defined in `stdlib.h`) to enable portability of the program.

7. If a function indicates an error with its return value, it *should* be checked in general. If the subsequent code depends on the successful execution of a function (e.g. resource allocation), then the return value *must* be checked.

8. Functions that do not take any parameters have to be declared with `void` in the signature, e.g., int get_random_int(void);.

9. Procedures (i.e., functions that do not return a value) have to be declared as `void`.

10. Error messages shall be written to `stderr` and should contain the program name `argv[0]`.

11. It is forbidden to use the functions: `gets`, `scanf`, `fscanf`, `atoi` and `atol` to avoid crashes due to invalid inputs.

FORBIDDEN	USE INSTEAD
gets	fgets
scanf	fgets, sscanf
fscanf	fgets, sscanf
atoi	strtol
atol	strtol

12. Documenation is mandatory. Format the documentation in Doxygen style (see Wiki and Doxygen's intro).

13. Write meaningful comments. For example, meaningful comments describe the algorithm, or why a particular solution has been chosen, if there seems to be an easier solution at a first glance. Avoid comments that just repeat the code itself
(e.g., i = i + 1; /* i is incremented by one */).

14. The documentation of a module must include: name of the module, name and student id of the author (`@author` tag), purpose of the module (`@brief`, `@details` tags) and creation date of the module (`@date` tag).

    Also the Makefile has to include a header, with author and program name at least.

15. Each function shall be documented either before the declaration or the implementation. It should include purpose (`@brief`, `@details` tags), description of parameters and return value (`@param`, `@return` tags) and description of global variables the function uses (`@details` tag).

    You should also document `static` functions (see `EXTRACT_STATIC` in the file `Doxyfile`). Document visible/exported functions in the header file and local (`static`) functions in the C file. Document variables, constants and types (especially `structs`) too.

16. Documentation, names of variables and constants shall be in English.

17. Internal functions shall be marked with the `static` qualifier and are not allowed to be exported (e.g., in a header file). Only functions that are used by other modules shall be declared in the header file.

18. All exercises shall be solved with functions of the C standard library. If a required function is not available in the standard library, you can use other (external) functions too. Avoid reinventing the wheel (e.g., re-implementation of `strcmp`).

19. Name of constants shall be written in upper case, names of variables in lower case (maybe with fist letter capital).

20. Use meaningful variable and constant names (e.g., also semaphores and shared memories).

21. Avoid using global variables as far as possible.

22. All boundaries shall be defined as constants (macros). Avoid arbitrary boundaries. If boundaries are necessary, treat its crossing.

23. Avoid side effects with `&&` and `||`, e.g., write if (b != 0) c = a/b; instead of if (b != 0 && c = a/b).

24. Each `switch` block must contain a `default` case. If the case is not reachable, write `assert(0)` to this case (defensive programming).

25. Logical values shall be treated with logical operators, numerical values with arithmetic operators (e.g., test 2 strings for equality by strcmp (...) == 0 instead of !strcmp (...)).

26. Indent your source code consistently (there are tools for that purpose, e.g., `indent`).

27. Avoid tricky arithmetic statements. Programs are written once, but read more times. Your program is not better if it is shorter!

28. For all I/O operations (read/write from/to `stdin`, `stdout`, files, sockets, pipes, etc.) use *either* standard I/O functions (`fdopen(3)`, `fopen(3)`, `fgets(3)`, etc.) *or* POSIX functions (`open(2)`, `read(2)`, `write(2)`, etc.). Remember, standard I/O functions are buffered. Mixing standard I/O functions and POSIX functions to access a common file descriptor can lead to undefined behaviour and is therefore forbidden.

29. If asked in the assignment, you must implement signal handling (`SIGINT`, `SIGTERM`). You must only use *async-signal-safe* functions in your signal handlers.

30. Close files, free dynamically allocated memory, and remove resources after usage.

31. Don't waste resources due to inconvenient programming. Header files shall not include implementation parts (exception: macros).

Exercise 1 Guidelines

Violation of following guidelines leads to a deduction of points in exercise 1.

1. Correct use of named sempahores (`sem_open(3)`, `sem_close(3)` `sem_unlink(3)`) and POSIX shared memory (`shm_overview(7)`) for inter-process communication of separated programs (e.g., server and client).

   Use your matriculation number as prefix in the names of all resources.

2. "Busy waiting" is forbidden. (Busy waiting is the repeated check of a condition in a loop for synchronization purposes.)

3. Synchronization with `sleep` is forbidden.