OPERATING SYSTEMS EXERCISE 2

Aufgabenstellung – encr

Implementieren Sie ein Programm, welches kontinuierlich Strings (Passwörter) von *stdin* einliest und diese von einem Kindprozess verschlüsseln lässt.

```
SYNOPSIS:
    encr
```

Jedes Mal, wenn der Elternprozess eine Zeile eingelesen hat, wird ein eigener Kindprozess erzeugt, der das Passwort (exklusive des abschließenden Newline-Zeichens) mittels *crypt(3)* verschlüsselt. Danach soll der Kindprozess mittels *sleep(3)* eine längere (pseudozufällige) Verarbeitungsdauer simulieren, um das nichtblockierende Einlesen des Elternprozesses zu demonstrieren, und schließlich das verschlüsselte Passwort auf *stdout* ausgeben.

Anleitung

Lesen Sie in einer Schleife zeilenweise Strings von *stdin* ein und entfernen Sie das abschließende Newline (`\n`). Erzeugen Sie mittels *fork(2)* einen Kindprozess und verschlüsseln Sie in diesem das Passwort; der Elternprozess liest zu diesem Zeitpunkt bereits wieder von *stdin* ein. Beachten Sie, dass in dieser Situation mehrere Kindprozesse asynchron und in nicht vorhersehbarer Reihenfolge terminieren können!

Der Exit-Status jedes Kindes muss so früh wie möglich abgeholt werden. Behandeln Sie das Signal *SIGCHLD*, um das zu gewährleisten.

Testen

Testen Sie ihr Programm z.B. wie folgt

```
$ ./encr
jamesbond
micky
encr: jamesbond -> aCqHsO4OYLnq.
encr: micky -> aCSHDy093k9FQ
turing07
encr: turing07 -> aCJvjJDrpvXWE
asdfqwer
encr: asdfqwer -> aCZpdSZB6ohMQ
```

Die Ausgabe ihres Programmes muss nicht genau wie in diesem Beispiel formatiert sein. Die Reihenfolge von Input und Output wird vermutlich anders sein, ebenso wie das verschlüsselte Wort je nach gewähltem Salt variiert.

Beachten Sie beim Testen auch den Fall, in dem der Elternprozess durch ein Signal terminiert wird, während noch Kindprozesse aktiv sind. Auch in diesem Fall muss der Elternprozess auf das Terminieren seiner Kinder warten.

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 `struct`s) 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 2 Guidelines

Violation of following guidelines leads to a deduction of points in exercise 2.

1. Correct use of fork/exec/pipes as tought in the lectures. For example, do not exploit inherited memory areas.
2. Ensure termination of child processes without `kill(2)` or `killpg(2)`. Collect the exit codes of child processes (`wait(2)`, `waitpid(2)`, `wait3(2)`).