OPERATING SYSTEMS EXERCISE 3

Aufgabenstellung – hangman

In dieser Aufgabe implementieren Sie das Spiel „Hangman“ in C. Die Implementierung soll aus zwei Programmen bestehen: einem Server, der die Geheimwörter angibt sowie den Spielstand verwaltet, und einem Client, mit welchem der Benutzer versuchen kann, Geheimwörter zu ermitteln. Die Interprozesskommunikation soll mittels Shared Memory realisiert werden und die Synchronisation mittels Semaphoren erfolgen.

Der Client soll dem Spieler ein Interface bieten, das den aktuellen Spielstand darstellt und dem Benutzer die Möglichkeit bietet, Buchstaben des Geheimwortes zu erraten. Der Server soll angeben, wie oft und an welcher Stelle des Geheimwortes der geratene Buchstabe vorkommt. Falls der Buchstabe im Geheimwort nicht enthalten ist, soll der Server einen weiteren Teil eines Galgens mit einem Gehängten zeichnen.

Es soll nicht zwischen Groß- und Kleinbuchstaben unterschieden werden. Wenn der Spieler (Client) nach 9 Fehlern das Geheimwort noch nicht erraten hat, wird der Galgen mit dem Gehängten komplett gezeichnet und der Spieler verliert.

Anleitung

Der Server muss eine beliebige Anzahl von Clients, die gleichzeitig und unabhängig voneinander spielen können, unterstützen. Die Kommunikation soll mittels einem einzigen Shared Memory Objekt erfolgen (**nicht** einem pro Client).

Server

```
USAGE: hangman-server [input_file]
```

Der Server legt zu Beginn die benötigten Ressourcen an. Falls eine Eingabedatei als Argument übergeben wird, liest der Server die Geheimwörter aus der Datei `input_file` ein. Beim Einlesen wird jede Zeile als ein Geheimwort interpretiert, wobei alle Zeichen außer ASCII Buchstaben und Leerzeichen ignoriert werden. Falls keine Eingabedatei angegeben wird, soll stattdessen von der Standardeingabe `stdin` eingelesen werden, bis `EOF` (`Ctrl+D`) auftritt.

Danach verwaltet der Server die Spiele der Clients und bedient auch deren Anfragen. Der Server muss dazu alle notwendigen Informationen für jeden Client verwalten. Insbesondere muss der Server alle Vorkommnisse der bisher korrekt geratenen Buchstaben und die Anzahl an Fehlern den Clients in jeder Runde bekannt geben.

Wenn das Spiel endet, soll der Server dem Client das Ergebnis übermitteln. Wenn ein Client ein neues Spiel starten will, soll der Server das Spiel mit einem bisher (für diesen Client) nicht verwendeten Geheimwort starten. Falls der Client ein Spiel mit allen Geheimwörtern gespielt hat, muss der Server den entsprechenden Client darüber informieren.

Wenn ein registrierter Client aus irgendwelchen Gründen (inklusive Signale) terminiert, soll der Server die entsprechenden Ressourcen freigeben. Zusatzlich muss der Server vor seiner eigenen Terminierung alle verwendeten Ressourcen (z.B., dynamisch reservierter Speicher, Semaphore, Shared Memory, usw.) freigeben. Der Server selbst soll durch die Signale `SIGINT` und `SIGTERM` „sauber“ terminierbar sein.

Client

```
USAGE: hangman-client
```

Der Client soll beim Starten zunächst versuchen sich zum Server zu verbinden („verbinden“ heißt hier zu überprüfen, ob die Semaphore, die der Server anlegt, tatsächlich existieren). Nach einem erfolgreichen Verbindungsaufbau registriert sich der Client beim Server und kann ein neues Spiel beginnen.

Der Client erlaubt nun dem Spieler in jedem Spielzug, einen bisher nicht gewählten Buchstaben auswählen. Falls das eingegebe Zeichen kein Buchstabe ist oder schon einmal geraten wurde, muss eine Fehlermeldung ausgegeben werden. Der Client schickt den ausgewählten Buchstaben an den Server, der den Spielzug durchführt und den Spielstand als Antwort bereit stellt.

Der nach jedem Spielzug ausgegebene Spielstand muss mindestens die folgenden Elemente enthalten:

- Das Geheimwort, wobei erratene Buchstaben auf ihren entsprechenden Positionen im Wort großgeschrieben dargestellt werden, während die restlichen Buchstaben durch '_' ersetzt werden. Leerzeichen werden von Anfang an dargestellt.
- Eine ASCII Zeichnung die ein anfaenglich unvollständiges Bild von einem Galgen mit Gehängten darstellt. Die Zeichnung muss nach jedem Fehler des Spielers erweitert und mit dem 9. Fehler vollständig werden.
- Eine Liste aller Buchstaben, die der Spieler bisher in diesem Spiel geraten hat.

Nach Ende eines Spiels wird zuerst eine entsprechende Nachricht und der aktuelle Win/Loss Stand (Anzahl der gewonnen und verlorenen Spiele) ausgegeben. Dann wird der Spieler gefragt, ob er mit dem nächsten Spiel weitermachen will, worauf der Spieler mit ja/nein (y/n) antworten soll. Falls er mit 'y' antwortet, wird auf ein neues Spiel vom Server gewartet, sonst soll der Client terminieren.

Falls der Server nach einem Spiel keine weiteren Spiele erstellen kann, also der Client ein Spiel mit jedem Geheimwort des Servers gespielt hat, wird der endgültige Win/Loss Stand ausgegeben und der Client beendet. Der Client kann jederzeit mit den Signalen `SIGINT` oder `SIGTERM` beendet werden. Außerdem soll der Client terminieren, wenn der Server beendet wurde.

Achten sie auf eine saubere Terminierung. Auch bei fehler- bzw. signalbezogener Terminierung sollen die Clients und den Server sauber terminieren und alle Ressourcen freigegeben werden.

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 3 Guidelines

Violation of following guidelines leads to a deduction of points in exercise 3.

1. It is forbidden to use the deprecated functions `gethostbyname`, `gethostbyaddr`, `gethostbyname2`, `gethostbyname_r`, `gethostbyname2_r` or any other function of that family documented on the man page `gethostbyname(3)`. Use `getaddrinfo(3)` instead.