OPERATING SYSTEMS EXERCISE 1

Aufgabenstellung B – Mastermind

Implementieren Sie einen Client und einen Server, die mittels TCP/IP miteinander kommunizieren. Dabei soll der Client den Spieler und der Server den Spielleiter des Spiels "Mastermind" implementieren. In diesem Spiel versucht der Spieler eine vom Spielleiter geheimgehaltene, geordnete Folge von Farben zu erraten. *Beachten Sie, dass der Spieler vollautomatisch agieren soll.* Daher ist die Implementierung einer Spielstrategie ebenfalls Teil der Aufgabe (siehe Abschnitt **??**).

In der hier verwendeten Variante des Spiels besteht eine Folge aus 5 Farben, wobei folgende 8 Farben zur Verfügung stehen: **b**eige, **d**unkelblau, **g**rün, **o**range, **r**ot, **s**chwarz, **v**iolett, **w**eiß.

Nachdem eine Verbindung zwischen dem Client und dem Server hergestellt wurde, wird sofort mit dem Spiel begonnen. Der Spieler sendet die von ihm vermutete korrekte Folge von Farben an den Spielleiter. Dieser antwortet mit der Anzahl korrekt positionierter Farben (die Anzahl roter Stifte im Brettspiel), und der Anzahl an Farben, die zusätzlich ebenfalls in der korrekten Lösung enthalten sind, jedoch an der falschen Position vermutet wurden (Anzahl *weißer* Stifte). Beachten Sie dabei, dass die Summe roter und weißer Stifte für eine Farbe die Kardinalität dieser Farbe in der korrekten Lösung nicht übersteigen kann.

Hierzu ein Beispiel: Angenommen die korrekte Lösung ist die Folge

```
rot, rot, grün, grün, grün
```

und der Spieler vermutet bei der korrekten Folge handle es sich um

```
rot, grün, rot, beige, rot
```

Dann ist lediglich die erste Farbe der vermuteten Lösung an der korrekten Position (ein roter Stift), und *zwei* weitere Elemente der Folge (einmal Farbe grün und *einmal* Farbe rot) sind zusätzlich an einer anderen Position in der korrekten Lösung enthalten.

Das Spiel endet, wenn der Spieler die korrekte Folge erraten hat, der Server einen Protokollfehler meldet, oder die maximale Anzahl an Runden (**35**) erreicht wurde. Am Ende des Spiels sollen sowohl Server als auch Client entweder die Anzahl gespielter Runden, oder den vom Server übermittelten Fehler ausgeben.

Implementierungshinweise

Server: Teile des Servers sind bereits vorgegeben. Bitte verwenden Sie dieses Template und erweitern Sie es entsprechend. Dem Server wird als erstes Argument der Port übergeben, auf dem er für die Clients erreichbar sein soll. Das zweite Argument ist ein String der Länge **5**, bestehend aus den Anfangsbuchstaben der 5 Farben der geheimen Zahlenfolge.

Der Server soll auf eingehende Verbindungen warten. Sobald eine Verbindung akzeptiert wurde, beginnt ein neues Spiel, und der Server beantwortet bis zum Ende des Spiels die Anfragen des Clients. Am Ende des Spiels werden entweder die Anzahl der gespielten Runden oder der zuletzt übermittelte Fehler ausgegeben, und das Serverprogramm wird mit einem entsprechenden Rückgabewert (siehe weiter unten) beendet. Sobald der Server eines der Signale *SIGINT* oder *SIGTERM* empfängt, soll der Serversocket geschlossen und das Programm mit Rückgabewert 0 beendet werden.

```
Server:
    SYNOPSIS
```

```
        server <server-port>  <secret-sequence>
    EXAMPLE
        server 1280 wwrgb
```

Client: Dem Client wird beim Aufruf der Hostname und die Portnummer des Servers übergeben (wenn der Client auf der selben Maschine wie der Server ausgeführt wird, dann geben Sie *localhost* als Hostnamen an). Legen Sie zuerst einen TCP/IP-Socket an. Stellen Sie dann die zum Hostnamen des Servers zugehörige IP-Adresse fest, und verbinden Sie sich mit dem Server. Danach wird sofort mit dem Spiel begonnen, und der Client übermittelt wiederholt die vermutete Farbfolge, bis der Server entweder kommuniziert, dass die Folge der Geheimfolge entspricht (Antwortfeld rot ist 5), oder einen Fehler übermittelt. Am Ende des Spiels werden entweder die Anzahl der gespielten Runden oder der zuletzt übermittelte Fehler ausgegeben. Danach soll der Socket geschlossen und das Programm beendet werden (Rückgabewert siehe nächster Absatz).

```
Client:
    SYNOPSIS
        client <server-hostname> <server-port>
    EXAMPLE
        client localhost 1280
```

Fehlermeldungen und Rückgabewerte: Ist eines der beiden Fehlerstatusbits (siehe Abschnitt **??**) gesetzt, sollen sowohl der Client als auch der Server terminieren. Bei einem Paritätsfehler (Bit 6 der Serverantwort gesetzt) soll die Meldung "Parity error" ausgegeben und beide Programme mit dem Wert 2 beendet werden. Bei einer Überschreitung der maximalen Rundenanzahl, wenn also der Spieler die Folge nicht erraten konnte (Bit 7 gesetzt), geben Sie die Meldung "Game lost" aus und beenden Sie beide Programme mit dem Exit-Code 3. Sind beide Bits gesetzt, sind beide Meldungen auszugeben und die Programme mit Rückgabewert 4 zu terminieren. Bei sonstigen Fehlern (z.B. ungültige Kommandozeilenargumente, Verbindungsfehler, ...) soll eine informative Fehlermeldung ausgegeben und mit Rückgabewert *EXIT_FAILURE* (1) terminiert werden. Alle Fehlermeldungen müssen auf *stderr* ausgegeben und von einem Zeilenumbruch gefolgt werden.

Endet das Spiel regulär (der Spieler errät die geheime Farbfolge), so soll die Anzahl gespielter Runden auf *stdout* ausgegeben werden (Rückgabewert 0).

Protokoll

Server und *Client* kommunizieren in Runden wie nachfolgend gezeigt. Der Client übermittelt pro Runde genau zwei Bytes, der Server genau ein Byte. Dabei muss im Falle des Clients stets zuerst das niedrigerwertige Byte (Bits 0–7), und dann das höherwertige Byte (Bits 8–15) übertragen werden, unabhängig von der Architektur der Zielplattform.

Client

Der *Client* schickt an den Server Nachrichten im folgenden Format:

[boxformatting=

,bitwidth=1.5em,endianness=big]16 0-15
1p 3color R 3color 3color 3color 3color L

Der Wert `color L` entspricht der Farbe ganz links am Spielbrett, der Wert `color_R` der Farbe ganz rechts. Den Farben werden die Werte entsprechend dem folgenden `enum` zugeordnet: `enum color {beige = 0, darkblue, green, orange, red, black, violet, white}`.

Der Wert `p` entspricht einem *Parity Bit* über die einzelnen Bits der Farben (Bit 0 bis Bit 14). Um dieses zu Berechnen, werden die einzelnen Bits mit einer *xor*-Operation verknüpft.

Server

Der *Server* empfängt die Nachricht des Clients und wertet den aktuellen Versuch des Clients aus. Der Wert `number red` entspricht der Anzahl an richtig erratenen Farben an der richtigen Position, `number white` der Anzahl an richtig erratenen Farben an der falschen Position (siehe Abschnitt **??**). Das Feld `status` hat folgende Bedeutung: Ist Bit 6 gesetzt, wurde das *Parity Bit* vom Client nicht richtig berechnet. Ein gesetztes Bit an Position 7 bedeutet dass die maximale Anzahl an Versuchen überschritten wurde (Sie haben verloren).

[boxformatting=

,bitwidth=2.2em,endianness=big]8 0-7
2status 3number white 3number red

Bewertung

Um das Beispiel erfolgreich zu lösen, müssen sowohl die Implementierung des Servers als auch des Clients korrekt funktionieren (siehe allgemeine Beispielanforderungen) und folgenden Anforderungen genügen:

Server: Teile des Servers sind bereits vorgegeben. Bitte verwenden Sie dieses Template und erweitern Sie es entsprechend. Der Server muss die korrekte Antwort auf gültige Anfragen übermitteln. Entspricht bei der 35ten Anfrage die vermutete Folge nicht der Geheimfolge, so muss das Bit 7 der Serverantwort gesetzt werden. Genau dann, wenn das Paritätsbit falsch berechnet wurde, ist das Bit 6 der Antwort zu setzen.

Client: Der Client muss jeweils innerhalb von einer Sekunde (auf dem Abgaberechner im TI-Labor) eine Anfrage an den Server generieren. Sendet der Server einen Fehlercode, so muss der Client sofort mit der entsprechenden Fehlermeldung terminieren.

Bonuspunkte: Für gute und ausgezeichnete Lösungen werden Bonuspunkte vergeben. Die Qualität einer Lösung wird dadurch bestimmt, wie viele Runden im Schnitt benötigt werden, um ein Spiel zu gewinnen (average rounds per game). Benötigt Ihre Lösung im Durchschnitt weniger als 20 Runden, und verliert kein einziges Spiel, so erhalten Sie 5 Bonuspunkte. Weitere 5 Bonuspunkte werden vergeben, wenn der Client zusätzlich jedes Spiel in durchschnittlich 8 oder weniger Runden gewinnt.

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.