OPERATING SYSTEMS BEISPIEL 1

Aufgabenstellung B – Battleship

Implementieren Sie einen Client und einen Server, die mittels TCP/IP miteinander kommunizieren. Dabei sollen der Client und der Server miteinander "Battleship" (deu.: "Schiffe versenken") spielen.

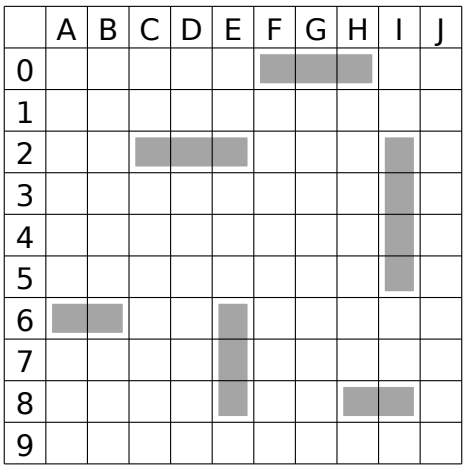
In diesem Spiel werden Schiffe auf einem Raster von 10x10 Kästchen platziert. Die Schiffe haben unterschiedliche Längen und nehmen daher unterschiedlich viele Kästchen ein. Schiffe dürfen nur horizontal oder vertikal platziert werden. Die Position der Schiffe wird vor dem Gegenspieler geheim gehalten, dessen Aufgabe es ist, die Schiffe zu versenken. Dafür kann der Gegenspieler in jedem Spielzug einen Schuss auf ein Kästchen abfeuern. Wurden alle Kästchen auf denen sich ein Schiff befindet getroffen, dann ist dieses Schiff versenkt.

Üblicherweise haben bei "Battleship" beide Spieler Schiffe und versuchen jeweils die des Gegners zu versenken. In dieser Aufgabe wird eine reduzierte Version gespielt, in der nur der Server Schiffe besitzt und der Client versucht diese zu versenken. *Beachten Sie, dass der Client vollautomatisch agieren soll.* Daher ist die Implementierung einer Spielstrategie ebenfalls Teil der Aufgabe (siehe Abschnitt Bewertung).

In dieser Variante des Spiels gibt es folgende Schiffe:

Bezeichnung	Anzahl	Länge
Schlachtschiff	1	4
Kreuzer	3	3
Zerstörer	2	2

Die Position der Schiffe wird dem Server durch Argumente auf der Kommandozeile übergeben. Die Anordnung der Schiffe könnte zum Beispiel folgendermaßen aussehen:



Nachdem eine Verbindung zwischen dem Client und dem Server hergestellt wurde, wird sofort mit dem Spiel begonnen. Der Client sendet die Koordinaten, auf die sein Schuss gerichtet ist und der Server teilt ihm mit, ob er damit ein Schiff getroffen hat, oder ob der Schuss ins Leere ging. Im oben abgebildeten Spielfeld würde beispielsweise ein Schuss auf **A4** ins Leere gehen, ein Schuss auf **D2** wäre ein Treffer.

Das Spiel endet, wenn der Client alle Schiffe versenkt hat, der Server einen Protokollfehler meldet, oder die maximale Anzahl an Runden (**80**) erreicht wurde.

Implementierungshinweise

> **Wichtig:** Beachten Sie, dass der wesentliche Teil in diesem Beispiel die korrekte Implementierung der Kommunikation zwischen Client und Server ist. Achten Sie auch darauf, dass Ihre Programme immer die richtigen Rückgabewerte verwenden!

Server: Teile des Servers sind bereits vorgegeben. Bitte verwenden Sie dieses Template und erweitern Sie es entsprechend. Dem Server kann als Option der Port übergeben werden, auf dem er für die Clients erreichbar sein soll. Außerdem werden dem Server 6 Argumente übergeben, die die Positionen der 6 Schiffe angeben. Jedes dieser Argumente hat Länge **4** und besteht aus den Koordinaten für Bug und Heck des Schiffes. So beschreibt zum Beispiel der String "`C2E2`" ein Schiff, welches die Kästchen **C2**, **D2** und **E2** einnimmt.

Der Server soll auf eingehende Verbindungen warten. Sobald eine Verbindung akzeptiert wurde, beginnt ein neues Spiel, und der Server antwortet bis zum Ende des Spiels auf die Anfragen des Clients. Sobald der Server eine Anfrage erhält, soll er zunächst die Korrektheit des Paritätsbit in der Nachricht des Clients überprüfen (siehe Abschnitt Protokoll). Sollte dieses falsch sein, antwortet der Server sofort mit dem entsprechenden Fehler-Status, ohne den Rest der Nachricht zu verarbeiten. Anschließend prüft der Server, ob die vom Client übermittelte Koordinate gültig ist und informiert den Client darüber, ob er mit seinem Schuss ein Schiff getroffen hat. Das Spiel endet sobald der Client alle Schiffe versenkt hat oder wenn am Ende der 80. Runde noch immer Schiffe übrig sind. Am Ende des Spiels soll die Verbindung geschlossen und das Programm beendet werden (Rückgabewerte siehe weiter unten).

```
Server:
    SYNOPSIS
        server [-p PORT] SHIP1...
    EXAMPLE
        server -p 1280 C2E2 F0H0 B6A6 E8E6 I2I5 H8I8
```

Client: Dem Client können beim Aufruf der Hostname und die Portnummer des Servers übergeben werden, ansonsten wird der Hostname per default auf *localhost* und der Port auf 1280 gesetzt. 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 Koordinaten, die durch seinen Schuss getroffen werden, bis der Server kommuniziert, dass alle Schiffe versenkt wurden, die maximale Rundenanzahl erreicht wurde oder einen Fehler übermittelt. Danach soll der Socket geschlossen und das Programm beendet werden (Rückgabewert siehe nächster Absatz).

```
Client:
    SYNOPSIS
        client [-h HOSTNAME] [-p PORT]
    EXAMPLE
        client -h localhost -p 1280
```

Fehlermeldungen und Rückgabewerte: Falls der Server in seiner Antwort einen Fehler anzeigt, sollen sowohl der Client als auch der Server terminieren. Bei einem Paritätsfehler sollen beide Programme die Meldung "Parity error" ausgeben und beide Programme mit dem Wert 2 beenden. Wenn der Server meldet, dass der Client einen ungültige Koordinate übertragen hat, geben Sie in beiden Programmen die Meldung "Invalid coordinate" aus und beenden Sie beide Programme mit dem Exit-Code 3. 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.

Das Spiel endet regulär wenn der Client alle Schiffe versenkt hat oder die maximale Rundenanzahl erreicht wurde. Gelingt es dem Client alle Schiffe zu versenken, so soll der Server die Anzahl gespielter Runden auf *stdout* ausgegeben und beide Programme mit Rückgabewert 0 beendet werden. Wenn es dem Client nicht gelingt in 80 Runden alle Schiffe zu versenken, dann sollen beide Programme die Meldung "Game lost" auf *stdout* ausgeben und ebenfalls mit Rückgabewert 0 terminieren.

Sobald der Server eines der Signale *SIGINT* oder *SIGTERM* empfängt, soll der Serversocket geschlossen und das Programm mit Rückgabewert 0 beendet werden.

1. Try SIGINT signal = CRTL + C command. If doesn't succeed:
2. Send a SIGTERM signal, this will stop the process but it will allow it to finish regularly and in most cases, it works. If not:
3. Send a SIGKILL signal, which is a brute-force signal that kills the process.

Note: There are more than 60 different 'kill' signals that you can use.
Also, you can use their numerical names:
SIGINT = 1
SIGTERM = 15
SIGKILL = 9

SYNTAX= kill -signal PID

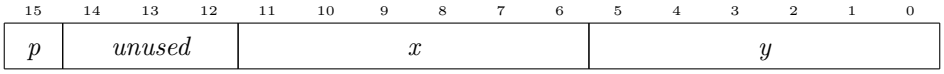
Protokoll

Wichtig: Achten Sie darauf, dass Sie das Protokoll exakt so implementieren, wie nachfolgend beschrieben! Die korrekte Implementierung des Protokolls wird beim Abgabegespräch überprüft, Abweichungen sind nicht zulässig. Insbesondere wird dabei auch festgestellt, ob Ihre Programme korrekt auf Fehler reagieren, also ob beispielsweise Ihr Server einen Paritätsfehler oder ungültige Koordinaten erkennt und ob Ihr Client auf Fehlermeldungen korrekt reagiert. Überlegen Sie sich geeignete Tests, um die Korrektheit Ihrer Implementierung sicherzustellen!

Server und *Client* kommunizieren in Runden wie nachfolgend gezeigt. Der Client übermittelt pro Runde genau zwei Bytes, der Server antwortet mit genau einem Byte.

Client

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



Die Werte `x` und `y` entsprechen dabei den Koordinaten des Kästchens, welches durch den Schuss des Client getroffen wird, wobei `x` der Spalten-Index (Spalte **A**: Index 0, **B**: 1, **C**: 2, ...) und `y` der Zeilen-Index des Kästchen ist. Beispielsweise entspricht das Kästchen **B5** den Indizes $\texttt{x} = 1$ und $\texttt{y} = 5$.

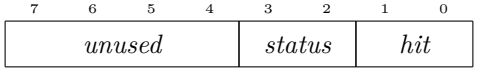
Der Wert `p` entspricht einem *Parity Bit*[1] über die restlichen Bits der Nachricht (Bit 0 bis Bit 14). Um dieses zu Berechnen, werden die einzelnen Bits mit einer *xor*-Operation verknüpft. Wir verwenden *even parity*, so dass die gesamte Nachricht immer aus einer geraden Anzahl an 1-Bits besteht.

Das Feld `unused` wird nicht verwendet, die entsprechenden Bits sollten auf 0 gesetzt werden.

Da die Nachricht aus mehr als einem Byte besteht, ist es wichtig die Byte-Reihenfolge[2] zu beachten: Der Client muss zuerst das Byte mit den niederwertigen Bits (Bit 0 bis Bit 7) und anschließend das Byte mit den höherwertigen Bits (Bit 8 bis Bit 15) übertragen.

Server

Der *Server* empfängt die Nachricht des Clients und überprüft, ob durch den Schuss eines seiner Schiffe getroffen wurde. Anschließend sendet der Server eine Antwort im folgenden Format:



[1] `https://en.wikipedia.org/wiki/Parity_bit`
[2] `https://de.wikipedia.org/wiki/Byte-Reihenfolge`

`hit` teilt dem Client mit, ob ein Schiff getroffen wurde und gibt auch Auskunft darüber, ob dieses versenkt wurde. Das Feld kann folgenden Werte annehmen:

`hit`	Bedeutung
0	Nichts getroffen (der Schuss ging ins Leere)
1	Ein Schiff wurde getroffen (aber nicht versenkt)
2	Ein Schiff wurde getroffen und damit versenkt, aber es gibt noch weitere Schiffe
3	Das letzte Schiff wurde versenkt (der Client gewinnt)

Das Feld `status` informiert den Client über den Spielstatus und eventuelle Fehler, die in seiner letzten Nachricht enthalten waren. Falls `status` einen Fehler anzeigt wird das Feld `hit` ignoriert und die entsprechenden Bits sollten auf 0 gesetzt werden. Folgende Werte können in `status` enthalten sein:

`status`	Bedeutung
0	Spiel läuft
1	Spiel beendet (Alle Schiffe versenkt oder maximale Rundenzahl erreicht)
2	Fehler: Die letzte Nachricht enthielt ein ungültiges Parity Bit
3	Fehler: Die letzte Nachricht enthielt eine ungültige Koordinate

Das Feld `unused` wird nicht verwendet, die entsprechenden Bits sollten auf 0 gesetzt werden.

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: Der Server muss nach dem Start die Argumente, die er von der Kommandozeile erhält, analysieren und falls ungültige Optionen oder Argumente übergeben wurde mit einer **usage**-Meldung terminieren. Dabei muss auch überprüft werden, ob die angegebenen Positionen der Schiffe gültig sind, also ob die entsprechenden Kästchen sich auf der Karte befinden und ob die Schiffe horizontal oder vertikal ausgerichtet sind. Anschließend muss der Server auf eine eingehende Verbindung warten. Falls ein alternativer Port als Option übergeben wird, hat der Server auf Verbindungen an diesem Port zu warten, ansonsten ist Port 1280 zu verwenden.

Sobald ein Client eine Verbindung zum Server hergestellt hat, beginnt das Spiel und der Server wartet auf Anfragen des Client. Während des Spiels muss der Server mitverfolgen welche seiner Schiffe an welchen Stellen getroffen wurden und den Client korrekt darüber informieren, wenn ein Schiff getroffen oder versenkt wurde und wenn alle seine Schiffe versenkt wurden. Falls der Client bei der Übermittlung ein falsches Paritätsbit oder ungültige Koordinaten sendet, oder es ihm nicht gelingt in 80 Zügen alle Schiffe zu versenken, hat der Server wie im Abschnitt Implementierungshinweise beschrieben darauf zu reagieren.

Client: Nach dem Start muss der Client die Argumente, die er von der Kommandozeile erhält, analysieren und falls ungültige Optionen übergeben werden mit einer **usage**-Meldung terminieren. Insbesondere dürfen die in Abschnitt Implementierungshinweise beschriebenen Optionen höchstens einmal übergeben werden und es dürfen im Anschluss an die Optionen keine weiteren Argumente mehr folgen. Anschließend muss der Client eine Verbindung zum Server unter dem angegebenen Hostnamen (oder **localhost** falls keine Hostname übergeben wurde) und dem angegebenen Port (oder 1280 falls kein Port übergeben wurde) herstellen.

Sobald der Client eine Verbindung zum Server hergestellt hat beginnt das Spiel und der Client sendet seine erste Anfrage. Sendet der Server einen Fehlercode, so muss der Client sofort mit der entsprechenden Fehlermeldung terminieren.

Bonuspunkte: Für gute und ausgezeichnete Lösungen werden Bonuspunkte vergeben. **Bonuspunkte werden nur vergeben, wenn der Client und der Server korrekt funktionieren.**

- **Server:** Sie erhalten 2 Bonuspunkte wenn Ihr Server überprüft, ob die korrekte Anzahl an Schiffen mit einer bestimmten Länge übergeben wurde (wie in Abschnitt Aufgabenstellung B – Battleship beschrieben), und ob zwischen den Schiffen jeweils mindestens ein Kästchen frei bleibt, also zwei Schiffe nie aneinander grenzen (auch nicht in diagonaler Richtung). Falls nicht, soll der Server mit einer **usage**-Meldung terminieren.
- **Client:** Zur Ermittlung der Bonuspunkte Ihres Client wird dieser mit dem vom Institut implementierten Server getestet und durchläuft dabei mehrere Test-Spiele. Die Tests bestehen ausschließlich aus gültigen Konfigurationen, also mit den in Abschnitt Aufgabenstellung B – Battleship angegebenen Schiffen, wobei zwei Schiffe nie aneinander grenzen (auch nicht in diagonaler Richtung). Sie erhalten 2 Bonuspunkte, wenn Ihr Client jedes Spiel gewinnt. Außerdem können Sie bis zu 6 weitere Bonuspunkte erhalten, wenn Ihr Client jedes Spiel gewinnt und dafür durchschnittlich besonders wenig Züge benötigt:

Durchschnittliche Anzahl an Zügen	< 70	< 55	< 40
Zusätzliche Bonuspunkte	2	4	6

Testfälle

Die folgenden Testfälle können Sie als Hilfestellung verwenden, um Ihre Implementierung zu testen. Außerdem sollen Sie Ihnen als Beispiel dienen für die Ausgaben, die von Ihren Programmen erwartet werden.

Fehlermeldungen beim Programmaufruf:

Server:

```
$ ./server -x G1I1 C5E5 A7B7 G6G8 I9J9 B0E0
./server: invalid option -- 'x'
usage: ./server [-p PORT] SHIP1...
$ ./server G1I1 C5E5 A7B77 G6G8 I9J9 B0E0
[./server] ERROR: wrong syntax for ship coordinates: A7B77
$ ./server G1I1 C5E5 A7Z7 G6G8 I9J9 B0E0
[./server] ERROR: coordinates outside of map: A7Z7
$ ./server G1I1 C5E5 A7B9 G6G8 I9J9 B0E0
[./server] ERROR: ships must be aligned either horizontally or vertically: A7B9
```

Client:

```
$ ./client -x
./client: invalid option -- 'x'
usage: ./client [-h HOSTNAME] [-p PORT]
$ ./client arg
usage: ./client [-h HOSTNAME] [-p PORT]
```

Fehlermeldungen während des Spieles:

Paritätsfehler:

```
$ ./server G1I1 C5E5 A7B7 G6G8 I9J9 B0E0 & sleep 1 && ./client
[./server] ERROR: parity error
[./client] ERROR: parity error
```

Ungültige Koordinate:

```
$ ./server G1I1 C5E5 A7B7 G6G8 I9J9 B0E0 & sleep 1 && ./client
[./server] ERROR: invalid coordinate
[./client] ERROR: invalid coordinate
```

Client verliert:

```
$ ./server G1I1 C5E5 A7B7 G6G8 I9J9 B0E0 & sleep 1 && ./client
[./server] game lost
[./client] game lost
```

Client gewinnt:

```
$ ./server G1I1 C5E5 A7B7 G6G8 I9J9 B0E0 & sleep 1 && ./client
[./server] client wins in 77 rounds
[./client] I win :)
```

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.
In other words, a violation of any of the following rules results in 0 points for your submission.

1. The program 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*. These flags must be used in the Makefile, of course. 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 must conform to the assignment. The program 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. The program 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*.

2. There must be a Makefile for the program implementing the targets: `all` to build the program 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

No additional guidelines, but note that correct argument parsing and sockets (1B) will be the main focus.