

OPERATING SYSTEMS EXERCISE 3

Aufgabenstellung – battleships

In dieser Aufgabe werden Sie eine Terminalversion des Spiels „Battleships“ (auch bekannt als „Schiffe versenken“) für 2 Spieler implementieren.

Das Spielfeld besteht aus 16 Feldern die in einem 4x4 Quadrat angeordnet sind.

Zu Beginn des Spiels müssen beide Spieler ein Schiff der Länge 3 auf seinem Spielfeld platzieren. Das Schiff kann horizontal, vertikal oder diagonal angeordnet werden.

Nachdem beide Spieler ihr Schiff platziert haben, beginnt das Spiel. In jeder Runde versuchen die Spieler das Schiff ihres Gegners zu treffen. Dazu geben sie eine Position am Spielfeld an, auf die sie schießen wollen. Nach jedem Schuss erhalten sie eine Rückmeldung, ob das Schiff des Gegners getroffen oder verfehlt wurde.

Ziel des Spiels ist es, das Schiff des Gegners so schnell wie möglich zu versenken. Ein Schiff gilt als versenkt, wenn alle 3 Felder des Schiffs getroffen wurden.

Anleitung

Das Spiel soll als Client/Server Programm realisiert werden, wobei ein Server immer nur ein Spiel zwischen 2 Clients unterstützen muss. Die Kommunikation zwischen Clients und Server soll per Shared Memory erfolgen.

Server

USAGE: battleships-server

Der Server soll ein Spiel zwischen 2 Spielern verwalten können. Sobald ein Spiel gestartet wird sollen die benötigten Ressourcen vom Server angelegt werden. Bei Beendigung eines Spiels müssen diese natürlich wieder freigegeben werden.

Zu Beginn wartet der Server bis sich 2 Spieler verbinden. Anschließend wird ein neues Spiel gestartet. Die beiden Clients müssen dem Server die Position ihrer Schiffe bekannt gegeben. Der Server ist für die Verwaltung der Position der Schiffe verantwortlich.

Danach beginnt das eigentliche Spiel, und der Server verarbeitet abwechselnd die Schüsse der Spieler. Stellen Sie sicher, dass der Server die Eingaben der Clients abwechselnd verarbeitet, also kein Spieler mehrere Züge direkt nacheinander durchführen kann. Nach jedem Schuss gibt der Server dem Client bekannt, ob das gegnerische Schiff getroffen oder verfehlt wurde. Sobald ein Spieler das Schiff des Gegners versenkt hat, teilt der Server das beiden Clients mit und beendet das Spiel.

Anschließend wartet der Server, bis sich wieder 2 Clients verbinden.

Falls der Server ein SIGTERM oder SIGINT Signal erhält, sollen alle angelegten Ressourcen (lokal angelegter Speicher, Shared Memory, Semaphoren, etc) korrekt freigegeben und terminiert werden.

Client

USAGE: `battelships-client`

Der Client ist für die Darstellung des Spielfelds und die Behandlung von Benutzereingaben zuständig.

Die Gestaltung der Oberfläche ist nicht weiter spezifiziert, außer dass mindestens das Spielfeld und eine kurze Anleitung mit den möglichen Aktionen angezeigt werden müssen.

Sobald der Client gestartet ist, verbindet er sich mit dem Server und wartet bis der Server ein neues Spiel startet.

Anschließend wird der Spieler aufgefordert, die Position seines Schiffs einzugeben.

Haben beide Spieler ihre Schiffe platziert, beginnt das Spiel.

Mögliche Aktionen sind:

- *Schuss auf Position am Spielfeld*: Angabe der Position am Spielfeld, auf die geschossen werden soll.
- *Aufgeben*: Der Client teilt dem Server mit, dass er aufgibt und terminiert. Daraufhin soll das Spiel beendet werden.

Am Spielfeld sollen die bisherigen Schüsse des Spielers angezeigt werden und ob es sich um einen Treffer handelt.

Der Client soll zusätzlich ordnungsgemäß terminieren sobald das Spiel gewonnen oder verloren ist, oder der Server (d.h. das Shared Memory oder die Semaphore) nicht mehr erreichbar ist.

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
<code>gets</code>	<code>fgets</code>
<code>scanf</code>	<code>fgets</code> , <code>sscanf</code>
<code>fscanf</code>	<code>fgets</code> , <code>sscanf</code>
<code>atoi</code>	<code>strtol</code>
<code>atol</code>	<code>strtol</code>

12. Documentation 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 first 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.