OPERATING SYSTEMS EXERCISE 1

Aufgabenstellung B – Coffee Maker

Am Institut für Technische Informatik wurde eine Kaffeemaschine mit einem Raspberry Pi mit Netzwerkanbindung und automatischer Kaffeekapsel-Versorgung erweitert. Nun soll ein Service "Kaffeeproduktion" entwickelt werden.

Anleitung

Implementieren Sie einen Client und einen Server, die mittels TCP/IP miteinander kommunizieren. Angestellte des Instituts starten den Client, um eine Anfrage an die Kaffeemaschine zu senden. Der Server soll auf der Kaffeemaschine Anfragen bearbeiten und die Kaffeeproduktion einleiten.

Nachdem eine Verbindung zwischen dem Client und dem Server hergestellt wurde, wird die Anfrage übermittelt. Angestellte können Tassengröße und Geschmacksrichtung des Kaffees wählen. Die Kaffeemaschine soll mit einem Status und gegebenenfalls der Produktionsdauer antworten.

Verwenden Sie geeignete Ausgaben (auf *stdout*) am Server und Client um Anfragen und Kaffeeproduktion folgen zu können. Alle Fehlermeldungen müssen auf *stderr* ausgegeben.

Server

Der Server soll auf eingehende Verbindungen warten. Sobald eine Verbindung akzeptiert wurde, wird die Anfrage bearbeitet. Neue Verbindungen werden unmittelbar nach Bearbeitung der Anfrage (d.h. lesen, berechnen, antworten) wieder zugelassen, d.h. der Server wartet nicht auf die Beendigung der Kaffeeproduktion (welche hier selbst nicht implementiert wird). Der Server sollte sich aber den Zeitstempel der Fertigstellung merken, um Anfragen während einer bereits laufenden Kaffeeproduktion korrekt bearbeiten zu können (Produktionsdauer).

Der Server verwaltet den aktuellen Status der Kaffeemaschine, welcher nach jeder Anfrage aktualisiert werden muss. Initial soll der Wassertank voll und die Kapselkammer, die gebrauchte Kapseln sammelt, leer sein. Pro Anfrage: soll je nach Tassengröße der Inhalt des Wassertanks verringert werden und die Anzahl der Kapseln in der Kapselkammer inkrementiert werden. Ist das Wasser für die aktuelle Anfrage nicht mehr ausreichend oder die Kapselkammer voll, soll ein Fehlercode an den Client zurück gesendet werden. Kann der Kaffee produziert werden, wird die Dauer der Produktion in Sekunden zurückgeschickt.

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

```
SYNOPSIS
    server [-p PORT] [-l WATER] [-c CUPS]
```

Die Größe des Wassertanks (default: 1l) und der Kapselkammer (default: 10 Kapseln) kann beim Starten des Servers ausgewählt werden. Die Portnummer ist per default 1821.

Client

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. Unmittelbar nach Verbindungsaufbau wird die Anfrage zur Kaffeeproduktion übermittelt und die Antwort des Servers ausgewertet und angezeigt. Danach soll der Socket geschlossen und das Programm beendet werden.

Der Client sollte bei erfolgreichem Verbindungsaufbau mit dem Server, sofort eine Antwort bekommen. Tritt ein Fehler auf (z.B.: keine Verbindung möglich), beendet der Client mit einem Fehlercode. Der Client sollte nie blockieren, Sie können daher auf eine Signalbehandlung im Client verzichten.

```
SYNOPSIS
    client [-h HOSTNAME] [-p PORT] SIZE FLAVOUR
```

Dem Client können Hostname (default: *localhost*) und Portnummer (default: *1821*) übergeben werden. Die Tassengröße (*SIZE*, ganzzahlig) und Geschmacksrichtung (*FLAVOUR*, C-String) des Kaffees sollen über die Argumente gesetzt werden.

Protokoll

Der Client übermittelt Tassengröße und Geschmacksrichtung. Folgende Werte sollen unterstützt werden:

Tassengröße: 50 - 330 ml

Geschmacksrichtung: {Decaffeinato, Kazaar, Volluto, Ciocattino, Vanilio, ...} - es sollen bis zu 32 Geschmacksrichtungen[1] unterstützt werden.

Die Anfrage soll in 2 Bytes übermittelt werden. Das letzte Bit soll ein Parity Bit[2] implementieren. Dieses ist am Server zu überprüfen.

Die Antwort des Servers ist von der Anfrage, vom Status der Kaffeemaschine (Wassertank, Kapselkammer) und der in Produktion befindlichen Kaffees abhängig. Je nach Status ergeben sich folgende Inhalte der Antwort-Nachricht.

Status: {OK - Kaffee kann produziert werden, NOK - fehlerhafte Anfrage oder Kaffeemaschine muss gewartet werden}

- **OK → Dauer der Kaffeeproduktion:** Dauer in Sekunden, die durch die Tassengröße (Wassermenge) bestimmt wird und Restdauer von Kaffeeproduktion. Pro 10ml wird 1s benötigt. Sollte die Kaffeeproduktion noch laufen, muss die Restdauer dazuaddiert werden. Runden Sie auf ganzzahlige Sekunden-Werte.
- **NOK → Fehlercode:** {Wassertank leer, Kapselkammer voll, ungültige Tassengröße, fehlerhaftes Parity Bit in der Anfrage}

Die Antwort soll in 2 Bytes übermittelt werden. Auch die Antwort soll ein Parity Bit enthalten und vom Client überprüft werden.

Definieren Sie ein geeignetes Format der Nachrichten (z.B.: IDs für die Geschmacksrichtungen vergeben). Stellen Sie sicher, dass die Übertragung unabhängig von der Architektur funktioniert.

[1] https://www.nespresso.com/at/de/grands-crus-uebersicht
[2] https://en.wikipedia.org/wiki/Parity_bit

Testen

```
$ ./client 100 Kazaar
[./client] Requesting a 100 ml cup of coffee of flavour 'Kazaar' (id=1).
[./client] connect - Connection refused

$ ./server -p 1234 &
[./server] Initial status: 1000ml water, 10 cups bin
[./server] Waiting for client...
```

Anforderung zweier Kaffees unmittelbar hintereinander:

```
$ ./client -h 127.0.0.1 -p 1234 100 Kazaar
[./client] Requesting a 100 ml cup of coffee of flavour 'Kazaar' (id=1).
[./server] Client connected.
[./server] New status: 900ml water, 1 cups bin
[./server] Finish in 10s.
[./server] Start coffee for 100ml cup with flavour 'Kazaar'.
[./server] Close connection to client.
[./server] Waiting for client...
[./client] Coffee ready in 10s.
$ ./client -h localhost -p 1234 100 Kazaar
[./client] Requesting a 100 ml cup of coffee of flavour 'Kazaar' (id=1).
[./server] Client connected.
[./server] New status: 800ml water, 2 cups bin
[./server] Another coffee still in production.
[./server] Finish in 19s.
[./server] Start coffee for 100ml cup with flavour 'Kazaar'.
[./server] Close connection to client.
[./server] Waiting for client...
[./client] Coffee ready in 19s.
```

Wenn man weiter Kaffee anfordert, bis der Wassertank leer bzw. die Kapselkammer voll ist:

```
:
$ ./client -p 1234 100 Kazaar
[./client] Requesting a 100 ml cup of coffee of flavour 'Kazaar' (id=1).
[./server] Client connected.
[./server] New status: 50ml water, 10 cups bin
[./server] Another coffee still in production.
[./server] Finish in 35s.
[./server] Start coffee for 100ml cup with flavour 'Kazaar'.
[./server] Close connection to client.
[./server] Waiting for client...
[./client] Coffee ready in 35s.
$ ./client -p 1234 100 Kazaar
[./client] Requesting a 100 ml cup of coffee of flavour 'Kazaar' (id=1).
[./server] Client connected.
[./server] Error - no_water_and_full_bin.
[./server] Close connection to client.
[./server] Waiting for client...
[./client] Error 3 - no_water_and_full_bin.
```

Bonus

In diesem Beispiel können Sie zusätzlich 10 Bonuspunkte erreichen, wenn Sie eine dynamische Liste für die Verwaltung der in der Produktion befindlichen Kaffees implementieren.

Im Zuge einer Anfrage wird die Liste aktualisiert: i) Beendete Produktionen werden von der Liste entfernt. ii) Ein neues Listenelement für die aktuelle Anfrage wird hinzugefügt. Der Server soll die aktuelle Liste nach jeder Anfrage ausgeben.

Definieren Sie eine geeignete Datenstruktur für ein Listenelement und implementieren Sie das dynamische Hinzufügen und Entfernen eines Elements.

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.