OPERATING SYSTEMS EXERCISE 3

Aufgabenstellung – banking

Es soll ein Programm geschrieben werden, das die Geldtransfers der Kunden verwaltet. Die Kunden können sich mit dem Programm verbinden und ihre Geldgeschäfte erledigen. Das Verwaltungsprogramm der Bank ist der Server und dieser kommuniziert mittels einem einzigen Shared Memory Objekts mit den Clients (**nicht** einem pro Client), die von den Kunden benutzt werden.

Die Clients können Geld abheben und einzahlen, es auf ein anderes Konto überweisen und den aktuellen Kontostand abfragen. Aus Gründen der Sicherheit und des Datenschutzes müssen alle Informationen beim Server bleiben und die Clients wissen nur ihre eigene Kontonummer. Außerdem darf pro Konto nur ein Client angemeldet sein und jeder Client soll sich auch abmelden, wenn er sich beendet.

Anleitung

Das Bank Account Management Tool soll als Client/Server Programm realisiert werden, wobei ein Server beliebig viele Clients bedient. Die Kommunikation zwischen Clients und Server soll per Shared Memory erfolgen.

Server

Der Server liest aus einer CSV-Datei die Liste der Konten ein. Jede Zeile ist ein Konto und besteht aus der Kontonummer, dem Trennzeichen `';'` und dem Kontostand. Wenn der Server beendet, gibt er den neuen Stand der Konten als CSV-Datei aus.

```
USAGE: bankserver [ACCOUNT_LIST [OUT_ACCOUNT_LIST]]
```

Zu Beginn liest der Server die Kontoliste ein und legt die benötigten Ressourcen an. Falls eine Datei als Argument übergeben wurde, wird die Liste aus dieser Datei gelesen, sonst wird die Liste von der Standardeingabe gelesen. Ein EOF kennzeichnet das Ende der Eingabe.

Der Server soll beliebig viele Clients gleichzeitig bedienen können. Bei den Anfragen, die von den Clients kommen, muss der Server zuvor prüfen, ob die Operationen gültig sind, bevor Änderungen am Kontostand gemacht werden. Ansonsten würde Geld verloren gehen und die Kunden wären unzufrieden.

Jeder Client muss sich vorher anmelden, bevor er Anfragen stellen kann. Außerdem darf nicht mehr als ein Client pro Konto angemeldet sein. Der Server muss dies überprüfen. Wenn sich ein Client vom Server trennt, muss sich dieser davor abmelden. Ansonsten gilt der Client als angemeldet und es kann sich kein weiterer Client für dieses Konto anmelden.

Der Server muss folgende Anfragen bearbeiten können: Abfragen des Kontostandes, Abheben von Geld, Einzahlen von Geld, Überweisen auf ein anderes Konto. Beim Abheben und Überweisen darf der Client danach keinen negativen Betrag am Konto haben, d.h. es muss davor geprüft werden, ob genug Geld am Konto ist. Alle Anfragen geben nur zurück, ob die Anfrage erfolgreich bearbeitet wurde oder nicht. Bei der Abfrage des Kontostandes hingegen wird auch der Betrag an den Client geschickt.

Beim Einlesen der Kontoliste darf der Kontostand negativ sein. Die Kontonummer besteht aus sieben Dezimalstellen und die Kontonummer 0 ist reserviert und darf von keinem Client benutzt werden. Für den Kontostand reicht eine vorzeichenbehaftete 32 bit große Zahl. Der Kontostand hat keine Nachkommastellen.

Wird das Signal `SIGUSR1` empfangen, soll die Datenbank, d.h. alle Konten mit aktuellem Kontostand, ausgegeben werden.

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

Client

Der Client bekommt als Argument eine Kontonummer übergeben und versucht sich mit dieser anzumelden. Schlägt die Anmeldung fehl, soll der Client mit `EXIT_FAILURE` terminieren.

```
USAGE: bankclient ACCOUNT_NR
```

Der eingeloggte Kunde kann anschließend folgende Abfragen vornehmen bzw. Kommandos ausführen:

- Kontostand ausgeben:
  `balance`
- Geld in der Höhe von `<amount>` abheben:
  `takeout <amount>`
- Geld in der Höhe von `<amount>` einzahlen:
  `payin <amount>`
- Geld in der Höhe von `<amount>` an das Konto `<target_account>` überweisen:
  `transfer <amount> <target_account>`

Das Ergebnis (`<result>` aus `"OK"` oder `"NOK"`) jeder Anfrage wird auf die Standardausgabe ausgegeben. Beim Abfragen des Kontostandes wird auch der Betrag (`<current_amount>`) ausgegeben. Folgendes Format soll dazu verwendet werden:

```
<result> [<current_amount>]
```

Beim Verbinden meldet sich der Client an und beim Trennen meldet er sich ab. Im Falle eines Fehlers soll sich der Client so gut es geht, beim Server abmelden. Der Client soll durch ein EOF an der Standardeingabe beendet werden.

Beispiele

Liste der Konten

Eine Beispiel-CSV-Datei als Input für den Server:

```
123456;0
1234;-1
98765;2000
45678;-2000
```

Aufrufe

Starten des Servers (Annahme: `test.csv` enthält obige Liste an Konten):

```
$ ./server test.csv
```

Starten eines Clients mit ungültiger Kontonummer:

```
$ ./client 54321
./client: Login failed.
$ echo $?
1
```

Starten eines Clients mit gültiger Kontonummer und Eingeben eines Kommandos:

```
$ ./client 1234
balance
OK -1
```

Weitere Befehle können zeilenweise übergeben werden:

```
takeout 100
NOK
payin 11
OK
balance
OK 10
transfer 20 45678
NOK
transfer 5 99
NOK
transfer 5 45678
OK
```

Anmeldung eines weiteren Clients zum selbigen Konto:

```
$ ./client 1234
./client: Login failed.
```

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.