OPERATING SYSTEMS BEISPIEL 3

Aufgabenstellung – auth

In dieser Aufgabe sollen Sie eine passwort-geschützte Datenbank entwickeln. Die Implementierung soll aus zwei Programmen bestehen: einem Server, der die Datenbank verwaltet und Anfragen über deren Inhalt bearbeitet, und einem Client, mit welchem der Benutzer Informationen der Datenbank vom Server setzen oder abfragen kann. Die Kommunikation zwischen den Prozessen soll mittels Shared Memory realisiert werden und die Synchronisation über Semaphore erfolgen.

Der Server speichert folgende Benutzerdaten - Name des Benutzers (`username`), Passwort (`password`) und ein Geheimnis (`secret`) - in einer Datenbank. Der Client soll dem Benutzer ein Interface bieten, mit dem der Benutzer diese Informationen vom Server setzen und abfragen kann.

Anleitung

Die Kommunikation zwischen den Clients und dem Server soll mittels einem einzigen Shared Memory Object erfolgen (**nicht** einem pro Client). Es darf auch nicht die gesamte Datenbank im Shared Memory geladen sein, sondern nur die Information, die der Server mit einem einzigen Client austauscht. Allerdings muss eine beliebige Anzahl von Clients gleichzeitig und unabhängig voneinander mit dem Server kommunizieren können.

Implementieren Sie ein geeinetes Protokoll, mit welchem die Clients und der Server über das Shared Memory Object kommunizieren. Überlegen Sie sich dafür eine Struktur, mit welcher Anfragen und Antworten im Shared Memory gespeichert werden. Legen Sie so viele Semaphore an, wie für die Synchronisierung der Kommunikation zwischen dem Server und den Clients benötigt werden. Achten Sie darauf Deadlocks zu vermeiden!

Server und Client sollen die Freigabe des Shared Memory Objects und der Semaphore koordinieren. Spätestens wenn der Server und alle Clients terminiert haben, müssen alle Semaphore und Shared Memory Objects freigegeben sein. Sollte der Server vor einigen der Clients terminieren, dann müssen auch alle verbleibenden Clients unverzüglich terminieren. Achten Sie deshalb auf eine geeignete und korrekte Signal- und Fehlerbehandlung.

Sobald der Server oder Client eines der Signale *SIGINT* oder *SIGTERM* empfängt, soll die aktuelle Transaktion beendet werden und das Programm anschließend mit dem Rückgabewert 0 beendet werden.

Server

```
USAGE: auth-server [-l database]
```

Der Server legt zu Beginn die benötigten Resourcen an. Anschließend lädt der Server die angegebene Datenbank in einer geeigneten Datenstruktur in den Speicher. Wird diese Option beim Start nicht angegeben, so verwenden Sie bitte eine leere Datenbank als Ausgangspunkt. Die Datenbank dient dem Server im Weiteren als Informationsquelle für das Bearbeiten der Client-Anfragen. Gibt es ein Problem beim Einlesen der Datenbank, so beenden Sie den Server mit einer Fehlermeldung.

Nach der Initialisierung, bearbeitet der Server Anfragen von Clients. Der Server legt den vom Client gewünschten Eintrag an oder sucht den entsprechenden Eintrag in der Datenbank und sendet dem Client eine Antwort mit der gewünschten Information.

Der Server soll bei *jeder Terminierung* seine Datenbank in eine Datei names `auth-server.db.csv` schreiben. Falls die Datei bereits existiert, soll diese einfach überschrieben werden.

Client

```
USAGE: auth-client { -r | -l } username password
```

Der Client wird entweder mit Option `-r` oder mit Option `-l` ausgeführt. Diese Optionen entsprechen den folgenden Ausführungsmodi:

Register Mit der Option `-r` kann ein neuer Benutzer registriert werden. Falls der Benutzer (`username`) schon existiert, soll der Server dies dem Client entsprechend kommunizieren. Der Client soll in diesem Fall nach Ausgabe einer entsprechenden Fehlermeldung terminieren (`EXIT_FAILURE`). Existiert noch kein Benutzer mit dem angegebenen Namen (`username`) in der Datenbank des Servers, so soll der Benutzer angelegt werden und dies dem Client mitteilen. Der Client gibt eine entsprechende Erfolgsmeldung aus und terminiert (`EXIT_SUCCESS`).

Login Mit der Option `-l` kann sich ein bereits registrierter Benutzer einloggen. Der Server soll überprüfen, ob der gegebene Benutzer im System vorhanden ist oder nicht. Falls der Benutzer ungültig ist, soll der Server dies dem Client kommunizieren. Der Client soll nach Ausgabe einer entsprechenden Fehlermeldung terminieren (`EXIT_FAILURE`). Im Erfolgsfall soll der Client eine entsprechende Erfolgsmeldung ausgeben und auf weitere Befehle des Benutzers warten.

Nach einem erfolgreichen Login soll der Benutzer über die Standardeingabe wiederholt einen der folgenden drei Befehle eingeben können:

write secret Der Benutzer soll eine Nachricht auf die Standardeingabe schreiben können, die als `secret` am Server gespeichert und mit dem eingeloggten Benutzer assoziiert wird.

read secret Der Client soll das entsprechende Geheimnis ausgeben.

logout Der Server loggt den Benutzer aus und der Client terminiert (`EXIT_SUCCESS`).

Der Client erfragt den gewünschten Befehl bis sich der Benutzer ausloggt. Gestalten Sie am Client ein einfaches Benutzerinterface, etwa:

```
Commands:
  1) write secret
  2) read secret
  3) logout
Please select a command (1-3):
```

Die Inputs (Befehlsnummer und Geheimnis) sollen von der Standardeingabe gelesen werden.

Um sicher zu gehen, dass der Client auch wirklich eingeloggt ist, soll bei erfolgreichem Login eine zufällige Session-ID vom Server erzeugt, und an den Client gesendet werden. Diese ID muss bei jeder Kommunikation vom Client an den Server gesendet werden, um zu verifizieren, dass es sich um den authentifizierten Client handelt. Sollte ein Client mit falscher Session-ID versuchen, zum Beispiel auf das *secret* eines anderen Clients zuzugreifen, so soll der Server eine Fehlermeldung ausgeben und dem Client signalisieren, dass die Session ungültig ist. Beim Logout soll die Session-ID zerstört werden.

Datenformat

Datenbank

Die Datenbank soll als CSV-Datei gestaltet werden. Jede Zeile beinhaltet ein Geheimnis `secret` eines Benutzers `username`, das mit einem Passwort `password` geschützt ist.

```
username;password;secret
```

Achten Sie darauf, keine unnötigen Leerzeichen zwischen den Semikolons einzufügen, und insbesondere darauf, beim Speichern des secrets keinen Zeilenumbruch (`'\n'`) mitzuspeichern, wodurch leere Zeilen in der Datenbank entstehen würden.

Sie dürfen für alle Felder (`username`, `password`, `secret`) eine maximale Länge definieren um Ihnen die Arbeit zu erleichtern.

Beispiel

```
Theodor;ilovemilka;I'm the best!
Anton;cforever;
Emil;osueisgreat;Something you would like to know, but I won't say ANYBODY!
```

Der zweite Benutzer namens `Anton` hat in diesem Beispiel kein Geheimnis gespeichert.

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 3 Guidelines

Violation of following guidelines leads to a deduction of points in exercise 3.

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.

Hints

Below are several hints regarding this exercise which should help you to ensure the guidelines.

Client/server architecture. Derive from the exercise if a client/server architecture makes sense. In case of a client/server architecture (server creates the resources):

- The client shall terminate with an appropriate error message if the resources cannot be found.
- Assume that more than one client can start concurrently. You can assume only one server runs at the same time.
- The server should stay functional after error-free termination of a client.

If you don't use a client/server architecture, assume that each process can be started once in random order.

Cleanup resources. Resources shall be cleaned up also in case of errors.

Note, shared memory objects and semaphores won't be automatically deleted after the termination of a program. Hence, these resources have to be explicitly removed.

Shared memory and semaphores, that haven't been removed due to a program error or crash, can be listed and removed with the usual commands, `ls` and `rm` respectively, in the folder `/dev/shm/`. You can identify your resources by the chosen name (matriculation number) or ownership of the resource files.

Avoid wasting resources. Use a shared memory of fixed size. Use a minimum number of semaphores possible, but ensure correct synchronization (you may ask our tutors during lab hours).

Termination. After correct program execution (without errors) a synchronous termination shall be performed, i.e., take care of the synchronization between running processes when deleting the resources.