OPERATING SYSTEMS EXERCISE 3

Aufgabenstellung – ProcDB

In dieser Aufgabe setzen Sie eine Prozessdatenbank in C um. Die Implementierung soll aus zwei Programmen bestehen: einem Server, der die Prozessdatenbank hält und Anfragen über deren Inhalt bearbeitet, und einem Client, mit welchem der Benutzer Informationen aus der Prozessdatenbank vom Server abfragen kann. Die Kommunikation zwischen den Prozessen soll mittels Shared Memory realisiert werden und die Synchronisierung über Semaphore erfolgen.

Der Client soll dem Benutzer ein Interface bieten, mit dem der Benutzer Informationen (*cpu*, *mem*, *time* oder *command*) über Prozesse vom Server abfragen kann. Jeder Prozess wird durch seine PID identifiziert. Weiters, kann jener Prozess abgefragt werden, bei welchem die gegebene Information den höchsten oder den niedrigsten Wert in der Datenbank hat. Auch die Summe und der Mittelwert bestimmter Informationen kann berechnet werden.

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 Prozessdatenbank 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. Insbesondere darf das Warten auf Input eines Clients nicht andere Clients blockieren.

Server und Client sollen Freigabe der Shared Memory Objects und Semaphore koordinieren. Spätestens wenn der Server und alle Clients terminiert haben, müssen alle Semaphore und Shared Memory Objects freigegeben sein.

Server

```
USAGE: procdb-server input-file
```

Der Server legt zu Beginn die benötigten Resourcen an. Anschließend liest er die als Argument die angegebene Eingabedatei ein, parst die Information (z.B. mittels `fgets`, `strtok`, `sscanf`) und legt sie in einer geeigneten Datenstruktur (z.B. Array, Liste, Hashtabelle) im Speicher ab. Diese dient dem Server im Weiteren als Informationsquelle für das Bearbeiten der Requests der Clients.

Anschließend bearbeitet der Server Anfragen von Clients (siehe „Client“). Der Server sucht den entsprechenden Eintrag in der Datenbank und sendet dem Client eine Antwort mit der gewünschten Information.

Wird das Signal `SIGUSR1` empfangen, soll die Datenbank, d.h. alle Daten zu den Prozessen (PID und Informationen), ausgegeben werden.

Der Server soll durch die Signale `SIGINT` und `SIGTERM` zum Terminieren gebracht werden. Dabei soll er „sauber“ beenden, also alle verwendeten Resourcen (z.B. dynamisch reservierter Speicher, Semaphore, Shared Memory) freigeben. Das soll auch im Fehlerfall passieren.

Client

`USAGE: procdb-client`

Der Client verbindet sich beim Starten zuerst mit dem Server („verbinden“ bedeutet hier zu überprüfen, ob die Semaphore und das Shared Memory Object, welche zur Kommunikation notwendig sind, existieren). Anschließend nimmt er vom Benutzer Befehle auf der Standardeingabe zum Abfragen von Informationen vom Server entgegen. Ein Befehl hat das (einzeilige) Format:

```
pid info
```

`pid` ist entweder die PID eines Prozesses, d.h. ein numerischer Wert, oder eine Zeichenkette aus {`"min"`, `"max"`, `"sum"`, `"avg"`}.

`info` ist eine Zeichenkette, die bestimmt welche Information zurückliefert werden soll:

- `"cpu"`: die CPU-Auslastung, ausgegeben in Prozent, auf Zehntelprozent (”0.1%”) genau.
- `"mem"`: der Speicherverbrauch, ausgegeben in Prozent, auf Zehntelprozent genau.
- `"time"`: die Zeitdauer, über welche der Prozess bereits läuft, ausgegeben in Stunden, Minuten und Sekunden
- `"command"`: das Kommando, mit welchem der Prozess gestartet wurde, als String.

Nur für die numerischen Informationen (`"cpu"`, `"mem"`, `"time"`) können anstatt einer konkreten PID oben genannte Zeichenketten (`"min"`, `"max"`, `"sum"`, `"avg"`) angegeben werden, welche den niedrigsten Wert, den höchsten Wert, die Summe, oder den Durchschnitt der gegebenen Information aller Einträge in der Datenbank abfragen.

Sobald der Client die Antwort erhält, gibt er die PID des Prozesses (oder `"-"` im Fall von Summe oder Durchschnitt, etc.) `pid` und den angeforderten Wert `value` aus. Zeichenketten sollen mit doppelten Hochkommas abgeschlossen ausgegeben werden. Folgendes Format ist zu verwenden:

```
pid value
```

Ungültige Abfragen sollen schon beim Client behandelt werden und für sie soll kein Request erzeugt werden. (Eine Anfrage für eine nicht existierende PID bildet keine ungültige Abfrage - dies muss der Server prüfen und eine entsprechende Antwort generieren.) Die Befehlseingabe endet mit *EOF* (Ctrl-D auf dem Terminal). Wie beim Server ist auch beim Client auf eine saubere Terminierung (auch bei Signalen und im Fehlerfall) zu achten.

Datenformat

Datenbank

Die Datenbank-Datei ist im CSV (comma-separated values) gegeben. Jeder Eintrag (Zeile) enthält die Information eines Prozesses, welcher aus folgenden, durch Komma getrennte Felder enthält:

- *pid*: Die Prozess ID.

- *cpu*: Die CPU-Auslastung in Promille (Zehntelprozent).
- *mem*: Der Speicherverbrauch in Promille.
- *time*: Die Laufzeit des Prozesses in Sekunden.
- *command*: Das Kommando als Zeichenkette, höchstens 56 Zeichen lang, mit doppelten Hochkommas abgeschlossen.

Beispiel:

```
2214,0,0,1392074,"/usr/bin/ssh-agent /usr/bin/dbus-launch --exit-with-sess"
2230,2,35,390837,"/usr/lib/thunderbird/thunderbird"
9270,0,0,1124541,"vim README.txt"
14141,24,66,89729,"/usr/lib/firefox/firefox"
```

Shared Memory

Es soll eine geeignete Struktur definiert werden, die Felder für die Anfrage, als auch für die Antwort enthält. Die Art der Information, die abgefragt wird (cpu, mem, time, command) soll als Aufzählungstyp definiert sein.

Beispiele

Starten des Servers (Annahme: `test.db` enthält obiges Datenbank-Beispiel):

```
$ ./server test.db
```

Starten des Clients und Eingeben eines Befehls:

```
$ ./client
2214 cpu
2214 0
```

Weitere Befehle können zeilenweise übergeben werden:

```
14141 mem
14141 6.6
sum cpu
- 2.6
min time
- 24h 55min 29s
9270 time
312h 22min 21s
2230 command
2230 "/usr/lib/thunderbird/thunderbird"
max mem
- 6.6
```

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.