OPERATING SYSTEMS EXERCISE 3

Aufgabenstellung – mrna

In diesem Beispiel soll einer der grundlegendsten Vorgänge in der Zelle, die Synthese von Proteinen (auch Translation genannt), anschaulich dargestellt werden.

Die *mRNA* (messenger RNA), auch Boten-RNA genannt, entspricht einem Teilabschnitt der DNA und wird aus dieser synthetisiert (auch Transkription genannt). Das daraus resultierende Template dient anschließend in der Zelle zur Produktion von unterschiedlichsten *Proteinen*.

Ein Protein besteht aus der Kombination von 20 verschiedenen *Aminosäuren* und die Information, welche Aminosäuren bei der Synthese aneinander gehängt werden müssen, werden in der mRNA gespeichert. Den dazu verwendeten Code nennt man auch den *genetischen Code*. (Siehe *Genetic Code*)

Die mRNA verwendet keinen binären Code, jedoch erinnert der Aufbau sehr an Konzepte aus der Informatik. Statt Bits [`0` oder `1`; Basis 2] werden *Nukleinbasen* [Adenin (`A`), Guanin (`G`), Cytosin (`C`) oder Uracil (`U`); Basis 4] zur Speicherung von Informationen verwendet. 8 Bits werden immer zu einem Byte zusammengefasst, 3 Nukleinbasen zu einem *Codon*. Dieses Codon kann nun einer der 20 Aminosäuren zugeordnet werden. Vergleichbar: ein Byte entspricht einem ASCII Zeichen in der ASCII-Tabelle. Da es aber nur 20 Aminosäuren gibt, jedoch $4^3 = 64$ mögliche Codons, sind einige Aminosäuren mehrfach belegt. Tabelle 1 zeigt die Zuordnung der Nukleinbasentripel zu den Aminosäuren.

Beispiel: UUU=Phe(F), UUC=Phe(F), AAA=Lys(K), AAG=Lys(K)

Eine Protein-Sequenz (die message) innerhalb einer mRNA-Sequenz beginnt immer mit einem *Start-Codon* (AUG) und endet mit einem *Stopp-Codon* (UAA, UAG oder UGA). Die dazwischen liegenden Codons geben an, aus welchen Aminosäuren das Protein schlussendlich synthetisiert werden soll. Ein Beispiel illustriert den Vorgang:

mRNA	... CACAGCCU AUG GUA GGG UCU GCU UAA CGCCGC ...
	AUG (START) — GUA=**V**, GGG=**G**, UCU=**S**, GCU=**A** — UAA (STOP)
Protein	**VGSA**

Anleitung

Die Implementierung besteht aus einem Server, der die mRNA Sequenz und Befehle entgegennimmt und entsprechend verarbeitet, und einem Client, der zur Eingabe von Daten dient und die Ergebnisse visualisiert. Wählen Sie eine geeignete maximale Länge der mRNA- und Protein-Sequenz. Die Kommunikation zwischen den Prozessen soll mittels Shared Memory realisiert werden und die Synchronisierung über Semaphore erfolgen.

Die Kommunikation zwischen den Clients und dem Server soll mittels einem einzigen Shared Memory Object erfolgen (**nicht** einem pro Client), welches nur die Information, die der Server mit einem einzigen Client austauscht enthalten darf. 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 Synchroni-

U	U	U	**Phe F**
		C	
		A	**Leu L**
		G	
	C	U	**Ser S**
		C	
		A	
		G	
	A	U	**Tyr Y**
		C	
		A	**STOP**
		G	
	G	U	**Cys C**
		C	
		A	**STOP**
		G	**Trp W**
C	U	U	**Leu L**
		C	
		A	
		G	
	C	U	**Pro P**
		C	
		A	
		G	
	A	U	**His H**
		C	
		A	**Gln Q**
		G	
	G	U	**Arg R**
		C	
		A	
		G	
A	U	U	**Ile I**
		C	
		A	
		G	**Met M START**
	C	U	**Thr T**
		C	
		A	
		G	
	A	U	**Asn N**
		C	
		A	**Lys K**
		G	
	G	U	**Ser S**
		C	
		A	**Arg R**
		G	
G	U	U	**Val V**
		C	
		A	
		G	
	C	U	**Ala A**
		C	
		A	
		G	
	A	U	**Asp D**
		C	
		A	**Glu E**
		G	
	G	U	**Gly G**
		C	
		A	
		G	

Tabelle 1: Genetischer Code: Dekodierung der Codons zu Aminosäuren

sierung 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.

Client

```
USAGE: mrna-client
```

Wenn der Client gestartet wird, soll dem Nutzer zuerst eine Liste aller verfügbaren Befehle ausgegeben werden:

```
$ ./mrna-client
Available commands:
 s - submit a new mRNA sequence
 n - show next protein sequence in active mRNA sequence
 r - reset active mRNA sequence
 q - close this client
```

Die Auswahl soll von der Standardeingabe gelesen werden.

`s` Dieser Befehl erlaubt es dem Nutzer eine neue mRNA-Sequenz an den Server zu senden. Der Server hinterlegt diese Sequenz in einem dem Client zugeordneten Datenspeicher. Die Eingabe wird durch zwei Leerzeilen beendet und es sollen alle ungültigen Zeichen und Leerraum (Tabs, Leerzeichen, Zeilenumbrüche) entfernt werden.

`n` Sollte für diesen Client eine mRNA-Sequenz am Server hinterlegt sein, gibt dieser Befehl das nächste Vorkommen einer Protein-Sequenz aus. Neben der gefundenen Protein-Sequenz selbst wird sowohl die Start als auch die End-Position in der betreffenden mRNA-Sequenz ausgegeben.

`r` Wenn das Ende einer aktiven mRNA-Sequenz erreicht wurde, kann der Nutzer mit diesem Befehl den Cursor für diesen Client zurückstellen. Dadurch kann wieder die gesammte mRNA-Sequenz mit dem n-Command durchlaufen werden.

`q` Dieser Befehl schließt den Client und gibt alle Ressourcen frei die der Client hält und die der Server für den Client hält (z.B. die aktive mRNA-Sequenz oder die Cursor-Position).

Server

```
USAGE: mrna-server
```

Der Server legt zu Beginn die benötigten Resourcen an. Der Server soll es erlauben, dass mehrere unabhängige Clients mit ihm gleichzeitig kommunizieren können. Es soll sichergestellt sein, dass jeder Client nur seine eigene mRNA-Sequenz sieht. Überlegen Sie sich dafür eine geeignete Strategie, mittels derer der Server die einzelnen Anfragen dem jeweiligen Client zuordnen kann.

Empfängt der Server einen n-Befehl, so bewegt er einen Cursor durch die aktive mRNA-Sequenz bis er ein Start-Codon (AUG) entdeckt. Nun wird ab dieser Position jedes Codon einer Aminosäure zugeordnet bis eines der Stopp-Codons (UAA, UAG, UGA) erreicht wird. Die gefundene Protein-Sequenz wird an den Client zurückgegeben. Die aktuelle Cursor-Position bleibt am Server gespeichert.

Beispiel

Folgendes Beispiel zeigt den Umfang und das zu verwendende Format der Ausgabe des Clients.

```
$ ./mrna-client
Available commands:
 s - submit a new mRNA sequence
 n - show next protein sequence in active mRNA sequence
 r - reset active mRNA sequence
 q - close this client
Enter new command. (Enter to end input): s
Enter new mRNA sequence. (Newline to end input):
CACAGCCUAUGGUAGGGUCUGCUUAACGCCGCUAUGUAUCAUAAGGAGACAUGACGCCGC

Enter new command. (Enter to end input): n
Protein sequence found [11/60] to [23/60]: MVGSA
Enter new command. (Enter to end input): n
Protein sequence found [36/60] to [51/60]: MYHKET
Enter new command. (Enter to end input): n
End reached [60/60], send r to reset.
```

```
Enter new command. (Enter to end input): r
Reset. [0/60]
Enter new command. (Enter to end input): n
Protein sequence found [11/60] to [23/60]: MVGSA
Enter new command. (Enter to end input): q
Close client.
```

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.