OPERATING SYSTEMS BEISPIEL 2

Aufgabenstellung – forksort

Schreiben Sie ein Programm, das Eingaben sortiert.

```
SYNOPSIS
    forksort
```

Anleitung

Das Programm soll die Daten von *stdin* lesen. Die erste Zeile muss eine Zahl größer Null sein, die die Anzahl der zu sortierenden Strings angibt. Die darauffolgenden Zeilen enthalten die Strings selbst; Sie können die Strings auf je 62 echte Zeichen begrenzen. (Vergessen Sie nicht dafür eine Konstante zu definieren!)

Beispiele

```
$ ./forksort
5
Heinrich
Anton
Theodor
Dora
Hugo
Anton
Dora
Heinrich
Hugo
Theodor
```

Arbeitsweise

Forksort funktioniert wie eine rekursive Merge-Sort-Variante[1].

1. Lesen Sie die Anzahl der Strings ein.
2. Ist die Anzahl gleich Eins, dann geben Sie das „sortierte“ Ergebnis aus (d.h. den Eingabe-String selbst).
3. Ist die Anzahl größer Eins, erstellen Sie zuerst vier Unnamed-Pipes (zwei je Kind) und erzeugen dann zwei Kindprozesse (nicht Kind- und Enkelkind) mittels *fork(2)*. Verwenden Sie *dup2(2)* und *execlp(3)* um *stdin*/*stdout* der neuen Prozesse auf eigene Streams umzuleiten und danach Forksort rekursiv aufzurufen.
4. Schreiben Sie je die Hälfte der Strings auf *stdin* der beiden Kinder im zuvor spezifiziertem Eingabeformat (d.h., Anzahl der Strings, Newline, die Eingabestrings zeilenweise). Ist die Anzahl der Strings ungerade, dann wählen Sie ein beliebiges Kind aus, das einen String mehr bekommt.

[1] Siehe insb. die graphisch aufbereitete Beispielausführung auf: http://de.wikipedia.org/wiki/Mergesort

5. Lesen Sie nun die Ausgabe der Kinder zeilenweise ein, und geben Sie diese aber nun "verschmolzen", d.h. sortiert zeilenweise aus (dieser Vorgang nennt sich *Merge*). Beachten Sie, dass die Ausgabe der Kinder bereits fuer sich sortiert ist. Sie können also solange die Strings des ersten Kindes ausgeben, bis der kleinste String des zweiten Kindes kleiner als der kleinste, noch nicht ausgegebene, String des ersten Kindes ist. Danach lesen Sie vom zweiten Kind, bis der String des ersten Kindes wiederum kleiner ist. Wiederholen Sie diesen Vorgang solange, bis alle Strings der Kinder abgearbeitet sind.

   Der Merge-Vorgang muss einen linearen Aufwand haben.

6. Verwenden Sie *wait(2)* oder *waitpid(2)*, um den Exit-Status der beiden Kinder zu lesen.

Hinweise

Achten Sie auf korrekte Terminierungsbedingungen von forksort um "Endlos-Rekursionen" zu vermeiden[2]. Dazu sind zwei Regeln zu beachten:

1. Forken Sie nur, wenn die Anzahl der zu sortierenden Strings größer eins ist.
2. Die an das jeweilige Kind übergebene Anzahl muss stets kleiner sein als die des Elternprozesses.

Beginnen Sie am besten, indem Sie eine Variante schreiben, die nur einen String sortieren kann. Erweitern Sie nun Ihr Programm für zwei Strings, indem Sie den beschriebenen Fork-Vorgang implementieren und jeweils einen String an jedes der beiden Kinder schreiben. Diese können ja schon einen einzelnen String sortieren. Ein einziger Aufruf von *strcmp(3)* als Merge-Implementation genügt um zu entscheiden welche Ausgabe der Kinder zuerst vom Elternprozess ausgegeben werden soll. Wenn das funktioniert, können Sie Ihre Merge-Implementierung verallgemeinern, um auch mehr als zwei Strings sortieren zu können.

Um Fehlermeldungen und Debug-Messages auszugeben, verwenden Sie stets *stderr*, da die *stdout* in den meisten Fällen umgeleitet ist.

Zum Testen der Merge-Implementierung können Sie mit *execlp(3)* statt Forksort auch `sort(1)` aufrufen. Für die endgültige Abgabe ist diese Vorgehensweise nicht gültig, zum Testen aber durchaus sinnvoll.

[2] http://en.wikipedia.org/wiki/Fork_bomb

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 2 Guidelines

Violation of following guidelines leads to a deduction of points in exercise 2.

1. Correct use of fork/exec/pipes as tought in the lectures. For example, do not exploit inherited memory areas.

2. Ensure termination of child processes without `kill(2)` or `killpg(2)`. Collect the exit codes of child processes (`wait(2)`, `waitpid(2)`, `wait3(2)`).