182.709 UE Betriebssysteme

EXERCISE 2

Last update 2022-11-17

Assignment – Integer Multiplication

Implement an algorithm for the efficient multiplication of large integers.

```
SYNOPSIS
    intmul
```

Instructions

The program takes two hexadecimal integers A and B with an equal number of digits as input, multiplies them and prints the result. The input is read from `stdin` and consists of two lines: the first line is the integer A and the second line is the integer B.

Your program must accept any number of digits. It must accept lowercase and uppercase characters as input. Terminate the program with exit status `EXIT_FAILURE` if an invalid input is encountered. If the number of digits is not a power of two or the two integers do not have equal length they should be filled by the correct number of leading zeroes.

The multiplication of the input values is calculated recursively, i.e. by the program calling itself:

1. If A and B both consist of only 1 hexadecimal digit, then multiply them, write the result to `stdout` and exit with status `EXIT_SUCCESS`.
2. Otherwise A and B both consist of $n > 1$ digits. Split them both into two parts each, with each part consisting of $n/2$ digits:

   A : | A_h | A_l | $\quad A = A_h \cdot 16^{n/2} + A_l$

   B : | B_h | B_l | $\quad B = B_h \cdot 16^{n/2} + B_l$

3. Using *fork(2)* and *execlp(3)*, recursively execute this program in four child processes, one for each of the multiplications $A_h \cdot B_h$, $A_h \cdot B_l$, $A_l \cdot B_h$ and $A_l \cdot B_l$. Use two unnamed pipes per child to redirect *stdin* and *stdout* (see *pipe(2)* and *dup2(3)*). Write the two values to be multiplied to *stdin* of each child. Read the respective result from each child's *stdout*. The four child processes must run simultaneously!
4. Use *wait(2)* or *waitpid(2)* to read the exit status of the children. Terminate the program with exit status `EXIT_FAILURE` if the exit status of any of the two child processes is not `EXIT_SUCCESS`.
5. The result of the multiplication $A \cdot B$ can now be calculated as follows:

   $$A \cdot B = A_h \cdot B_h \cdot 16^n + A_h \cdot B_l \cdot 16^{n/2} + A_l \cdot B_h \cdot 16^{n/2} + A_l \cdot B_l$$

   Find a clever way to write the result of this operation to `stdout`, even if the numbers are too large for the C data types. Remember that your program must deal with integers of any size! Leading zeros must also be printed such that the result has twice as many digits as the longer input value. The output must use lowercase characters. Terminate the program with exit status `EXIT_SUCCESS`.

Hints

- Think of a way to add the four intermediate results together one digit at a time while keeping track of the carry.
- In order to avoid endless recursion[1], fork only if the input number is greater than 1.
- To output error messages and debug messages, always use *stderr* because *stdout* is redirected in most cases.

Examples

```
$ cat 1.txt
3
3
$ ./intmul < 1.txt
09
```

```
$ cat 2.txt
1A
B3
$ ./intmul < 2.txt
122e
```

```
$ cat 3.txt
1000
0001
$ ./intmul < 3.txt
00001000
```

```
$ cat 4.txt
13A5D87E85412E5F
7812C53B014D5DF8
$ ./intmul < 4.txt
09372e47ae47c3f68e45d1a816906f08
```

Mandatory testcases

Input shown in blue color. Output to *stdout* shown in black. (Note that in the following output sections `EXIT_SUCCESS` equals `0`, and `EXIT_FAILURE` equals `1`. Refer to `stdlib.h` for further details.)

Testcase 1: easy-1

```
$>echo -e "3\n4" | ./intmul
0c
$>echo $?
0
```

Testcase 2: easy-2

```
$>echo -e "3\n3" | ./intmul
09
$>echo $?
0
```

Testcase 3: easy-3

```
$>echo -e "1A\nb3" | ./intmul
122e
$>echo $?
0
```

Testcase 4: easy-4

```
$>echo -e "1000\n0001" | ./intmul
00001000
$>echo $?
0
```

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

Testcase 5: easy-5

```
$>echo -e "aB\n7" | ./intmul
04ad
$>echo $?
0
```

Testcase 6: easy-6

```
$>echo -e "Deadbe\nef" | ./intmul
00000000cfe43462
$>echo $?
0
```

Testcase 7: easy-7

```
$>echo -e "13A5D87E85412E5F\n7812C53B014D5DF8" | ./intmul
09372e47ae47c3f68e45d1a816906f08
$>echo $?
0
```

Testcase 8: large-numbers-1

```
$>( for i in `seq 1 6`; do echo "A$i" | md5sum | cut -d " " -f1 | tr -d \\n; [ $i = 3 ] && echo ""; \
    done; echo ""; ) > testfilelarge1
$>cat testfilelarge1
fe6975a937016c20b43b17540e6c6246cc5e9216b37b3f82dfe398ed311d16e63a0666c07bad1a857d8359a70dedb685
dbd432ede8ea89ff7a24d6d7b47513c5c9dc7db2c2a84ccf1176c37e292185f7b79dafd9f1b32a27598872bba8a8f5da
$>./intmul < testfilelarge1
0000000000000000000000000000000000000000000000000000000000000000da77199a3a45673ef39f7d105bfe5d0c70d367ca8 \
    05a2137462f6b409ab581c062fde13267f74ec31aa43a2cb5ac24c2fb5016810a8e0297ca979a4785c003c77eb0c3aad68d1 \
    7f7dcb16c35521dfcf4268cdb23d260740f48ffe9846b62b642
$>echo $?
0
```

Testcase 9: invalid-input-1

```
$>echo -e "1" | ./intmul
$>echo $?
1
```

Testcase 10: invalid-input-2

```
$>echo -e "1\nx" | ./intmul
$>echo $?
1
```

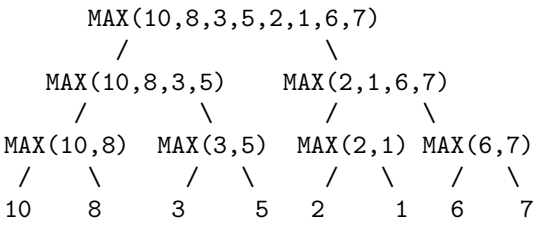
Testcase 11: invalid-input-3

```
$>echo -e "2\n2.0" | ./intmul
$>echo $?
1
```

Bonus exercise, 5 points

Print the parent and child relations in form of a tree to *stdout*. Print all children which are forked from the parent. The tree should be readable at least to a depth of three. For every node, the intermediate result should be printed.
The tree must only be printed if the compile flag `-DOSUETREE` is set.
Depending on how often you call fork the wider the tree becomes. A simple tree example which searches for the maximum number in a set is shown below.

```
10
```



Instructions on how to print the tree

- leaf node:
  A leaf node should print the substep executed by it to *stdout* with a terminating newline.
- inner node:
  - To get the necessary identation use several blank characters. Think about a good way to find the right number of blank characters. For example you could use precalculated values or calculate the number from the first line you read from the children.
  - Calculate the intermediate result and print this and the executed operation to *stdout*.
  - Slash and backslash, which represent the branches of the tree, are printed to *stdout*.
  - Read the output from the children line by line via a pipe. This means read the first line from the first child, then the first line from the second child and so on. Remove the newline characters. Line up the results and then print it with a terminating newline to *stdout*. Do this for each line returned by the child.

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 exactly to the assignment. The program(s) shall operate according to the specification/assignment given the test cases in the respective assignment. Additional white spaces or any other deviation from the specified input and output format may lead to a failure of the respective test case.

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).

32. To comply with the given testcases, the program output must exactly match the given specification. Therefore you are only allowed to to print any debug information if the compile flag `-DDEBUG` is set.

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)`).