Problem Sheet 11

Henri Sota
h.sota@jacobs-university.de
Computer Science 2022

November 29, 2019

Problem 11.1

a)
```c
#include <unistd.h>

int main(int argc, char *argv[]) {
    for (; argc > 1; argc--) {
        if (0 == fork()) {
            (void) fork();
        }
    }
    return 0;
}
```

Number of processes created for the following invocations of the program:

1)
```
./foo
```

Calling foo with no arguments ($argc = 1$) only creates one process and no child processes, because parent(root of all others) process doesn't enter the loop. Number of child processes: 0

2)
```
./foo a
```

Calling foo with 1 argument ($argc = 2$) creates 3 processes and 2 of those are child processes. Parent process enters the loop and the fork() in the conditional statement creates a child process that behaves the same way as the parent process after the fork() call. Parent process doesn't pass the if-conditional because return value of fork() is the PID of its child. Meanwhile the child process passes the if-conditional statement because the return value of fork() on itself is 0 (denoting that this isn't the parent). Inside the if-conditional body another fork() is found. This creates a child process of the child process of the parent(root) process. All 3 processes do not enter the next iteration of the loop as *argc* now has been decremented to 1. Number of child processes: 2

3)
```
./foo a b
```

Calling foo with 2 arguments ($argc = 3$) creates 9 processes and 8 of those are child processes. The first time the loop iterates is the same as foo being called with 1 argument ($argc = 2$). Currently we have 3 processes, 2 of which are child processes. All 3 processes enter the next iteration of the loop as *argc* has been decremented to 2 which allows for the body of the for-loop to run again. Each of these 3 processes acts as a parent(root process) of a child process which passes the if-conditional and forks itself again to create a child of itself. So each one has 2 processes as parts of its tree. In total there are $3 + 2 * 3 = 9$ processes of which 8 are child processes and the one left is the parent(root of all processes). Number of child processes: 8

4)
```
./foo a b c
```

Calling foo with 3 arguments ($argc = 4$) creates 27 processes and 26 of those are child processes. The first 2 times the loop iterates is the same as foo being called with 2 arguments ($argc = 3$). Currently we have 9 processes, 8 of which are child processes. All 9 processes enter the next iteration of the loop as $argc$ has been decremented to 2 which allows for the body of the for-loop to run again. Each of these 9 processes acts as a parent(root process) of a child process which passes the if-conditional and forks itself again to create a child of itself. So each one has 2 processes as parts of its tree. In total there are $9 + 2 * 9 = 27$ processes of which 26 are child processes and the one left is the parent(root of all processes). Number of child processes: 26

5)
```
./foo a b c d
```

Calling foo with 4 arguments ($argc = 5$) creates 81 processes and 80 of those are child processes. The first 3 times the loop iterates is the same as foo being called with 3 arguments ($argc = 4$). Currently we have 27 processes, 26 of which are child processes. All 27 processes enter the next iteration of the loop as $argc$ has been decremented to 2 which allows for the body of the for-loop to run again. Each of these 27 processes acts as a parent(root process) of a child process which passes the if-conditional and forks itself again to create a child of itself. So each one has 2 processes as parts of its tree. In total there are $27 + 2 * 27 = 81$ processes of which 80 are child processes and the one left is the parent(root of all processes). Number of child processes: 80

Formula to calculate the number of processes based on the number of arguments:

$$f(n) = \text{number of processes} \qquad n = \text{number of arguments}$$

$$\begin{aligned} f(0) &= 1 \\ f(1) &= 3 \\ f(2) &= 9 \\ &\vdots \\ f(n) &= f(n-1) + 2f(n-1) = 3f(n-1) \end{aligned}$$

$$f(n) = 3^n$$

From this, this number of child processes in the root process tree is: $f(n) = 3^n - 1$

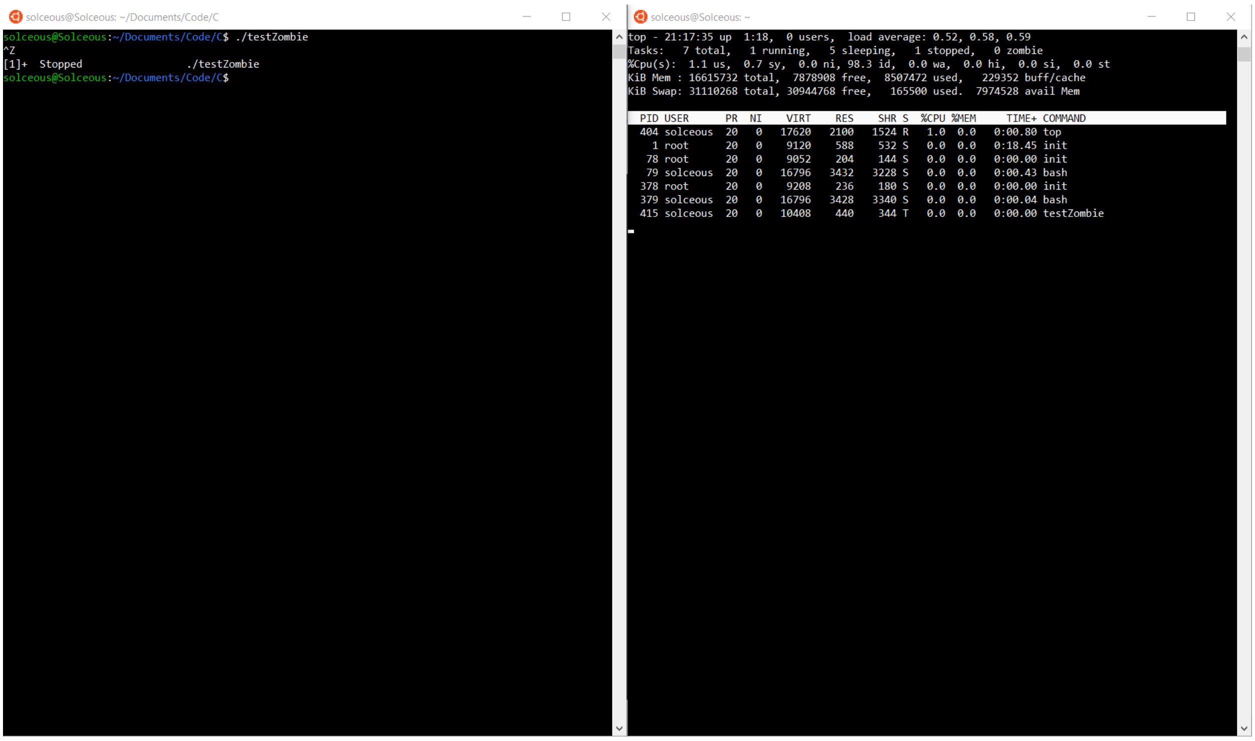
b) Write a C program to create n zombie processes based on the number of command line arguments program:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char *argv[])  {
    for (; argc > 1; argc--) {
        printf("Parent awake!\n");
        pid_t pid = fork();
        // Check if process is parent
        if (pid > 0) {
            printf("Parent put to sleep!\n");
            // Make parent sleep for 1 second therefore prolonging
            // the wait() call of the parent to check on the child and
            // remove the child process entry from the process table
            sleep(1);
        } else if (pid == 0) {
            // Child process will exit therefore terminated before the
            // parent process was able to reap the child using wait()
            printf("Child process turning into zombie!\n");
            exit(0);
            // After exiting, our child process becomes a zombie process
        }
    }
    sleep(20); // Place program to sleep in order to stop execution
    return 0;
}
```

Each time the loop body is iterated over, fork() is called on the parent process (root). Using the if-conditional, the parent process is put to sleep for 1 second and the child process is terminated therefore making the parent not able to reap the child process entry from the process table and turning the child process into a zombie. n zombie processes will be created using this program, depending on the number of command line arguments the executable file has been called with.

Documentation has been captured after zombie processes were created and the parent process had been put to sleep in order to capture processes using top utility in the terminal. Number of zombies can be seen on the right side, second row showing the tasks running. Zombie processes are characterized by the fact that they don't use up any system resources except a very tiny amount of system memory to store its process descriptor. We can therefore check a zombie process by checking the columns *VIRT*, *RES*, *SHR* on the right side.

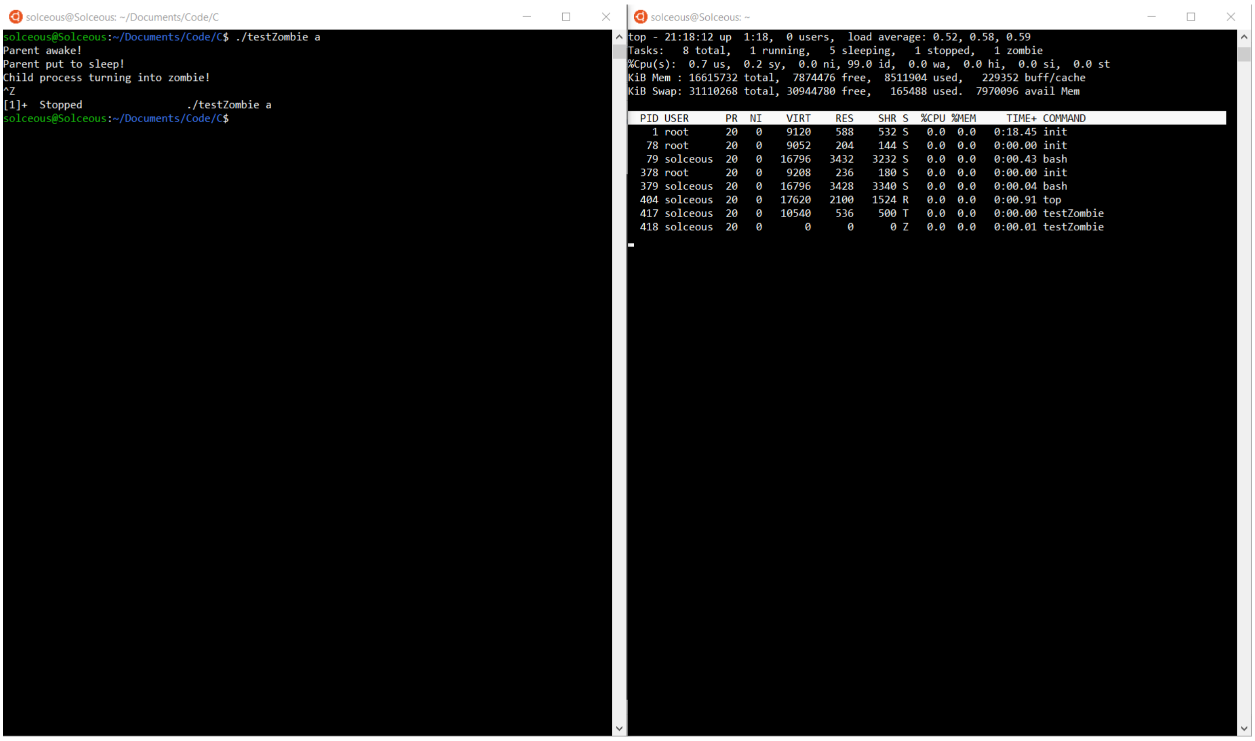


```
solceous@Solceous:~/Documents/Code/C$ ./testZombie
^Z
[1]+  Stopped                 ./testZombie
solceous@Solceous:~/Documents/Code/C$
```

```
top - 21:17:35 up  1:18,  0 users,  load average: 0.52, 0.58, 0.59
Tasks:   7 total,   1 running,   5 sleeping,   1 stopped,   0 zombie
%Cpu(s):  1.1 us,  0.7 sy,  0.0 ni, 98.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 16615732 total,  7878908 free,  8507472 used,   229352 buff/cache
KiB Swap: 31110268 total, 30944768 free,   165500 used.  7974528 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
  404 solceous  20   0   17620   2100   1524 R   1.0  0.0   0:00.80 top
    1 root      20   0    9120    588    532 S   0.0  0.0   0:18.45 init
   78 root      20   0    9052    204    144 S   0.0  0.0   0:00.00 init
   79 solceous  20   0   16796   3432   3228 S   0.0  0.0   0:00.43 bash
  378 root      20   0    9208    236    180 S   0.0  0.0   0:00.00 init
  379 solceous  20   0   16796   3428   3340 S   0.0  0.0   0:00.04 bash
  415 solceous  20   0   10408    440    344 T   0.0  0.0   0:00.00 testZombie
```

Running program with 0 command line arguments creates only one process, the parent one.

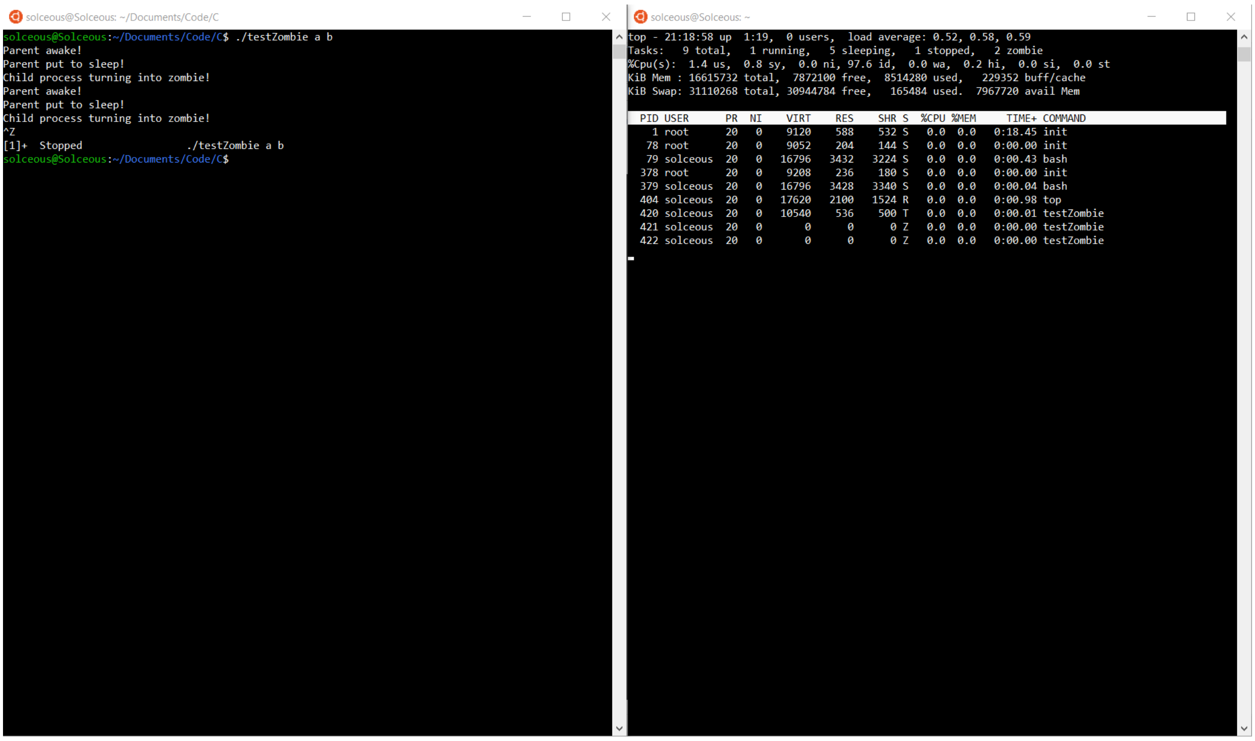


```
solceous@Solceous:~/Documents/Code/C$ ./testZombie a
Parent awake!
Parent put to sleep!
Child process turning into zombie!
^Z
[1]+  Stopped                 ./testZombie a
solceous@Solceous:~/Documents/Code/C$
```

```
top - 21:18:12 up  1:18,  0 users,  load average: 0.52, 0.58, 0.59
Tasks:   8 total,   1 running,   5 sleeping,   1 stopped,   1 zombie
%Cpu(s):  0.7 us,  0.2 sy,  0.0 ni, 99.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 16615732 total,  7874476 free,  8511904 used,   229352 buff/cache
KiB Swap: 31110268 total, 30944780 free,   165488 used.  7970096 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
    1 root      20   0    9120    588    532 S   0.0  0.0   0:18.45 init
   78 root      20   0    9052    204    144 S   0.0  0.0   0:00.00 init
   79 solceous  20   0   16796   3432   3232 S   0.0  0.0   0:00.43 bash
  378 root      20   0    9208    236    180 S   0.0  0.0   0:00.00 init
  379 solceous  20   0   16796   3428   3340 S   0.0  0.0   0:00.04 bash
  404 solceous  20   0   17620   2100   1524 R   0.0  0.0   0:00.91 top
  417 solceous  20   0   10540    536    500 T   0.0  0.0   0:00.00 testZombie
  418 solceous  20   0       0      0      0 Z   0.0  0.0   0:00.01 testZombie
```

Running program with 1 command line argument creates 2 processes, the parent one and the child which becomes a zombie process.

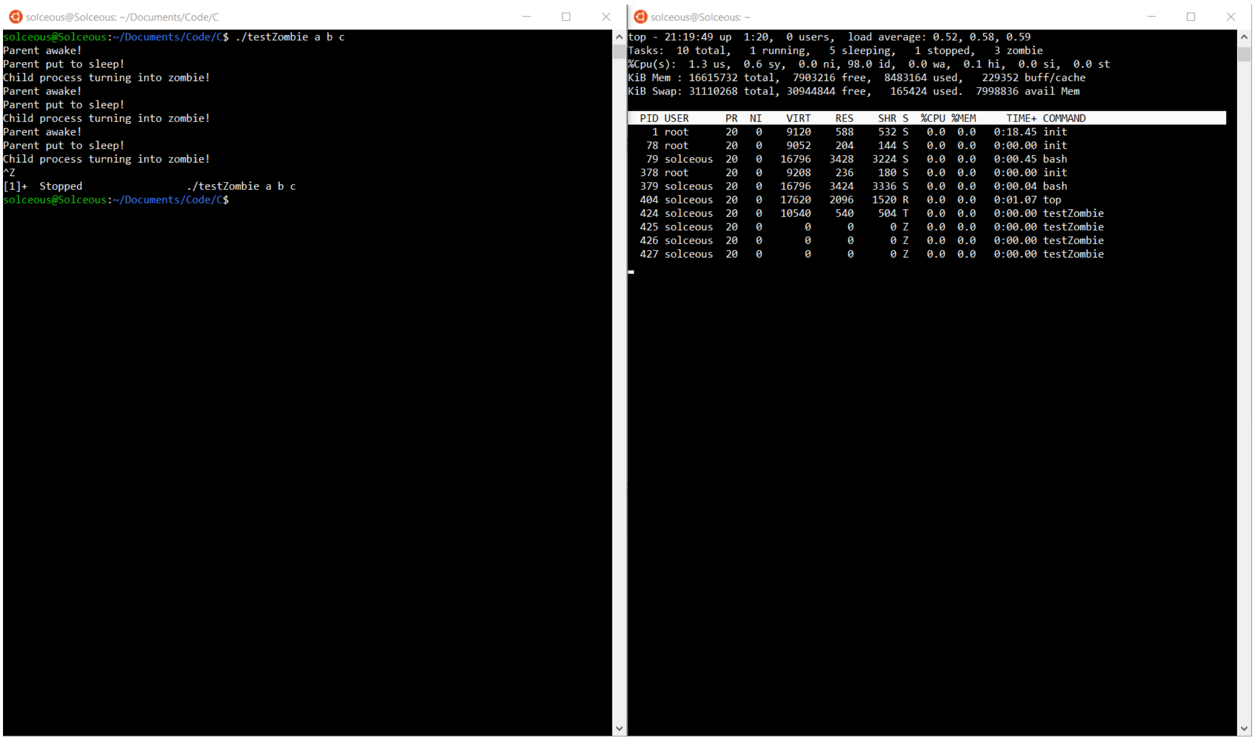


```
solceous@Solceous:~/Documents/Code/C$ ./testZombie a b
Parent awake!
Parent put to sleep!
Child process turning into zombie!
Parent awake!
Parent put to sleep!
Child process turning into zombie!
^Z
[1]+  Stopped                 ./testZombie a b
solceous@Solceous:~/Documents/Code/C$
```

```
top - 21:18:58 up  1:19,  0 users,  load average: 0.52, 0.58, 0.59
Tasks:   9 total,   1 running,   5 sleeping,   1 stopped,   2 zombie
%Cpu(s):  1.4 us,  0.8 sy,  0.0 ni, 97.6 id,  0.0 wa,  0.2 hi,  0.0 si,  0.0 st
KiB Mem : 16615732 total,  7872100 free,  8514280 used,   229352 buff/cache
KiB Swap: 31110268 total, 30944784 free,   165484 used.  7967720 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
    1 root      20   0    9120    588    532 S   0.0  0.0   0:18.45 init
   78 root      20   0    9052    204    144 S   0.0  0.0   0:00.00 init
   79 solceous  20   0   16796   3432   3224 S   0.0  0.0   0:00.43 bash
  378 root      20   0    9208    236    180 S   0.0  0.0   0:00.00 init
  379 solceous  20   0   16796   3428   3340 S   0.0  0.0   0:00.04 bash
  404 solceous  20   0   17620   2100   1524 R   0.0  0.0   0:00.98 top
  420 solceous  20   0   10540    536    500 T   0.0  0.0   0:00.01 testZombie
  421 solceous  20   0       0      0      0 Z   0.0  0.0   0:00.00 testZombie
  422 solceous  20   0       0      0      0 Z   0.0  0.0   0:00.00 testZombie
```

Running program with 2 command line arguments creates 3 processes, the parent one and 2 child processes which becomes zombie processes.

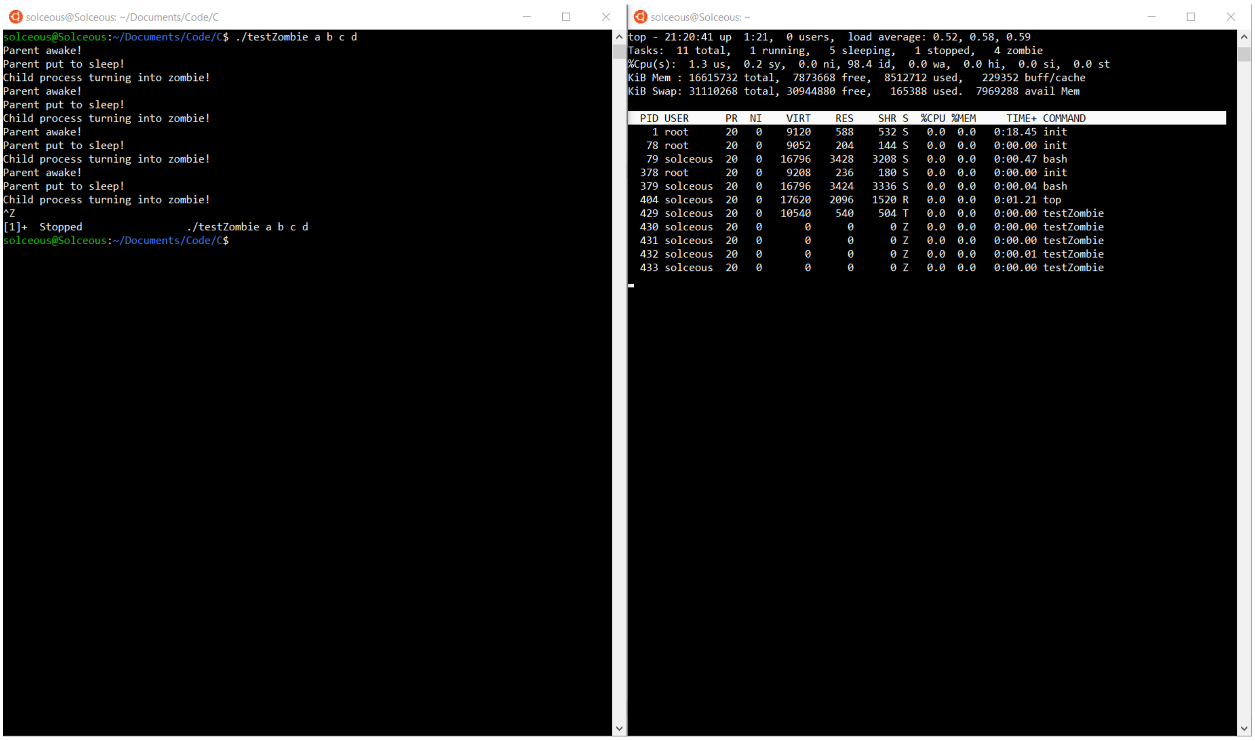


```
solceous@Solceous:~/Documents/Code/C$ ./testZombie a b c
Parent awake!
Parent put to sleep!
Child process turning into zombie!
Parent awake!
Parent put to sleep!
Child process turning into zombie!
Parent awake!
Parent put to sleep!
Child process turning into zombie!
^Z
[1]+  Stopped                 ./testZombie a b c
solceous@Solceous:~/Documents/Code/C$
```

```
top - 21:19:49 up  1:20,  0 users,  load average: 0.52, 0.58, 0.59
Tasks:  10 total,   1 running,   5 sleeping,   1 stopped,   3 zombie
%Cpu(s):  1.3 us,  0.6 sy,  0.0 ni, 98.0 id,  0.0 wa,  0.1 hi,  0.0 si,  0.0 st
KiB Mem : 16615732 total,  7903216 free,  8483164 used,   229352 buff/cache
KiB Swap: 31110268 total, 30944844 free,   165424 used.  7998836 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
    1 root      20   0    9120    588    532 S   0.0  0.0   0:18.45 init
   78 root      20   0    9052    204    144 S   0.0  0.0   0:00.00 init
   79 solceous  20   0   16796   3428   3224 S   0.0  0.0   0:00.45 bash
  378 root      20   0    9208    236    180 S   0.0  0.0   0:00.00 init
  379 solceous  20   0   16796   3424   3336 S   0.0  0.0   0:00.04 bash
  404 solceous  20   0   17620   2096   1520 R   0.0  0.0   0:01.07 top
  424 solceous  20   0   10540    540    504 T   0.0  0.0   0:00.00 testZombie
  425 solceous  20   0       0      0      0 Z   0.0  0.0   0:00.00 testZombie
  426 solceous  20   0       0      0      0 Z   0.0  0.0   0:00.00 testZombie
  427 solceous  20   0       0      0      0 Z   0.0  0.0   0:00.00 testZombie
```

Running program with 3 command line arguments creates 4 processes, the parent one and 3 child processes which becomes zombie processes.



```
solceous@Solceous:~/Documents/Code/C$ ./testZombie a b c d
Parent awake!
Parent put to sleep!
Child process turning into zombie!
Parent awake!
Parent put to sleep!
Child process turning into zombie!
Parent awake!
Parent put to sleep!
Child process turning into zombie!
Parent awake!
Parent put to sleep!
Child process turning into zombie!
^Z
[1]+  Stopped                 ./testZombie a b c d
solceous@Solceous:~/Documents/Code/C$
```

```
top - 21:20:41 up  1:21,  0 users,  load average: 0.52, 0.58, 0.59
Tasks:  11 total,   1 running,   5 sleeping,   1 stopped,   4 zombie
%Cpu(s):  1.3 us,  0.2 sy,  0.0 ni, 98.4 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 16615732 total,  7873668 free,  8512712 used,   229352 buff/cache
KiB Swap: 31110268 total, 30944880 free,   165388 used.  7969288 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
    1 root      20   0    9120    588    532 S   0.0  0.0   0:18.45 init
   78 root      20   0    9052    204    144 S   0.0  0.0   0:00.00 init
   79 solceous  20   0   16796   3428   3208 S   0.0  0.0   0:00.47 bash
  378 root      20   0    9208    236    180 S   0.0  0.0   0:00.00 init
  379 solceous  20   0   16796   3424   3336 S   0.0  0.0   0:00.04 bash
  404 solceous  20   0   17620   2096   1520 R   0.0  0.0   0:01.21 top
  429 solceous  20   0   10540    540    504 T   0.0  0.0   0:00.00 testZombie
  430 solceous  20   0       0      0      0 Z   0.0  0.0   0:00.00 testZombie
  431 solceous  20   0       0      0      0 Z   0.0  0.0   0:00.00 testZombie
  432 solceous  20   0       0      0      0 Z   0.0  0.0   0:00.01 testZombie
  433 solceous  20   0       0      0      0 Z   0.0  0.0   0:00.00 testZombie
```

Running program with 4 command line arguments creates 5 processes, the parent one and 4 child processes which becomes zombie processes.

Problem 11.2

a) Given syntax rules in BNF

```
<expression> ::= <term> | <expression> "+" <term>
<term> ::= <factor> | <term> "*" <factor>
<factor> ::= <constant> | <variable> | "(" <expression> ")"
<variable> ::= "x" | "y" | "z"
<constant> ::= <digit> | <digit> <constant>
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Expression to reduce: $4 * (2 * x + 7)$
Converting to terminal symbols: ”4” ”*” ”(” ”2” ”*” ”x” ”+” ”7” ”)”
Reduction to the start symbol of the grammar applying one rule each step to the leftmost symbol:

```
"4" "*" "(" "2" "*" "x" "+" "7" ")"
<digit> "*" "(" "2" "*" "x" "+" "7" ")"
<constant> "*" "(" "2" "*" "x" "+" "7" ")"
<factor> "*" "(" "2" "*" "x" "+" "7" ")"
<term> "*" "(" "2" "*" "x" "+" "7" ")"
<term> "*" "(" <digit> "*" "x" "+" "7" ")"
<term> "*" "(" <constant> "*" "x" "+" "7" ")"
<term> "*" "(" <factor> "*" "x" "+" "7" ")"
<term> "*" "(" <term> "*" "x" "+" "7" ")"
<term> "*" "(" <term> "*" <variable> "+" "7" ")"
<term> "*" "(" <term> "*" <factor> "+" "7" ")"
<term> "*" "(" <term> "+" "7" ")"
<term> "*" "(" <expression> "+" "7" ")"
<term> "*" "(" <expression> "+" <digit> ")"
<term> "*" "(" <expression> "+" <constant> ")"
<term> "*" "(" <expression> "+" <factor> ")"
<term> "*" "(" <expression> "+" <term> ")"
<term> "*" "(" <expression> "+" <term> ")"
<term> "*" "(" <expression> ")"
<term> "*" <factor>
<term>
<expression>
```

Found the non-terminal starting symbol which is <expression> by reduction.

b) In this particular case, there is no additional information needed to reduce our expression $4 * (2 * x + 7)$ in a meaningful way, because in the end, after reduction, the expression was replaced by the start symbol of the grammar, <expression>. Having parentheses as part of <factor> is decisive as the expression can be deduced to another form which is not equivalent to the one we ended up with, in case that the parentheses weren't there.

Problem 11.3

a) Distance module

```haskell
-- a) Code ed
module Distance (ed) where

-- Polymorphic function implements the Levenshtein Distance string metric
-- It takes two lists and returns an Int representing the Levenshtein Distance
-- It pattern matches the base cases:
--      both lists empty    => distance = 0
--      one list empty      => distance = length of the non-empty list
-- It guards cases when:
--      both first elements are equal   => distance = lev (tail firstList) (
    tail secondList)
--      all other cases => distance = 1 + minimum of [lev (list with appended
    first element of second list to first list) (secondList),
--                                                    lev (tail firstList) (
    secondList),
--                                                    lev (firstList with
    replaced first element with secondList’s first element) (secondList)]

ed :: Eq a => [a] -> [a] -> Int
ed [] [] = 0
ed [] ys = length ys
ed xs [] = length xs
ed (x:xs) (y:ys)
    | x == y    = ed xs ys
    | otherwise = 1 + minimum [ed (y:(x:xs)) (y:ys), ed xs (y:ys), ed (y:xs) (
    y:ys)]
```

b) Unit test for Distance module

```haskell
-- b) Code Unit tests for Distance Module
module Main where

    import Distance
    import Test.HUnit

    tests = TestList [
                TestCase (assertEqual "" 0 $ ed "" ""),
                TestCase (assertEqual "" 8 $ ed "" "checkbox"),
                TestCase (assertEqual "" 10 $ ed "lumberjack" ""),
                TestCase (assertEqual "" 1 $ ed "throw" "throws"),
                TestCase (assertEqual "" 3 $ ed "throw" "throwing"),
                TestCase (assertEqual "" 9 $ ed "kickboxing" "oxygenize"),
                TestCase (assertEqual "" 9 $ ed "oxygenize" "kickboxing"),
                TestCase (assertEqual "" 10 $ ed "kickboxing" "oxygenizes"),
                TestCase (assertEqual "" 3 $ ed [1.0,3.4,5.2,4.3] [ 1.0]),
                TestCase (assertEqual "" 2 $ ed [1,2,3] [1,2,3,4,5]),
                TestCase (assertEqual "" 4 $ ed [1,2,8,9] [1,2,8,9,3,9,2,3]),
                TestCase (assertEqual "" 3 $ ed [’k’,’i’,’t’,’t’,’e’,’n’] [’s
    ’,’i’,’t’,’t’,’i’,’n’,’g’])
            ]

    main = runTestTT tests
```

Unit tests are written to check for these cases:

- Edit distance between any list and itself is 0.
- No two lists have a negative edit distance.

- For an arbitrary list x, if you apply exactly one change to it, producing y, the edit distance between x and y should be 1.
- Given two lists x and y, compute the distance d between them. Then, change y, yielding y', and compute its distance from x: it should differ from d by at most 1.
- After applying n edits to a list x, the distance between the edited list and x should be at most n.
- The function should be symmetric: the edit distance from list x to y should be the same as from list y to x.