Introduction to Computer Science
Jacobs University Bremen
Dr. Jürgen Schönwälder

Course: CH08-320101
Date: 2017-11-21
Due: 2017-11-28

ICS Problem Sheet #10

Problem 10.1: *fold function duality* (2+2+2 = 6 points)

The fold functions compute a value form a list by applying an operator to the list elements and by using a neutral element. The foldl function assumes that the operator is left associative, the foldr function assumes that the operatore is right associative. For example, the function call

```
foldl (+) 0 [3,5,2,1]
```

results in the computation of ((((0+3)+5)+2)+1) and the function call

```
foldr (+) 0 [3,5,2,1]
```

results in the computation of (3+(5+(2+(1+0)))). The value computed by the fold functions may be more complex than a simple scalar. It is very well possible to construct a new list as part of the fold. For example:

```
map' :: (a -> b) -> [a] -> [b]
map' f xs = foldr (\x acc -> f x : acc) [] xs
```

The evaluation of `map' (+3) [1,2,3]` results in the list `[4,5,6]`. There are several duality theorems that can be stated for the fold functions. Proof the following three duality theorems:

a) Let `op` be an associative operation with `e` as the neutral element. Then the following holds for finite lists `xs`:

```
foldr op e xs = foldl op e xs
```

b) Let `op1` and `op2` be two operations for which

```
x `op1` (y `op2` z) = (x `op1` y) `op2` z
x `op1` e           = e `op2` x
```

holds. Then the following holds for finite lists `xs`:

```
foldr op1 e xs = foldl op2 e xs
```

c) Let `op` be an associative operation and `xs` a finite list. Then

```
foldr op a xs = foldl op' a (reverse xs)
```

holds with

```
x op' y = y op x
```

Problem 10.2: *fork system call* (1+3 = 4 points)

Consider the following C program (let me call the source file `happy-fork.c`.)

```
#include <unistd.h>

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

a) Assume the program has been compiled into `happy-fork` and that all system calls succeed at runtime. How many child processes are created for the following invocations of the program:

   - `./happy-fork`
   - `./happy-fork a`
   - `./happy-fork a b`
   - `./happy-fork a b c`
   - `./happy-fork a b c d`

b) Write a Linux assembly program (x86 64-bit Linux) using the GNU assembler that does the same as the C program shown above. It should in addition print `x\n` right before exiting. The assembly code should invoke the system calls directly, do not use any library calls. See `hello-asm-syscall.s` in the lecture notes as a starting point. Compile your assembly code with `gcc -nostdlib -static`. Make sure your assembly code has proper comments so that we can understand it.

   The `fork()` system call number is 57 on Linux. You can find the value of argc at `(%rsp)` at `_start`.