Introduction to Computer Science
Jacobs University Bremen
Dr. Jürgen Schönwälder

Module: CH-232
Date: 2021-11-19
Due: 2021-11-26

ICS 2021 Problem Sheet #11

Problem 11.1: *fork system call* (2+3 = 5 points)

Consider the following C program:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static void action(int m, int n)
{
    printf("(%d,%d)\n", m, n);
    if (n > 0) {
        if (fork() == 0) {
            action(m, n-1);
            exit(0);
        }
    }
}

int main(int argc, char *argv[])
{
    for (int i = 1; i < argc; i++) {
        int a = atoi(argv[i]);
        action(a, a);
    }
    return 0;
}
```

a) Assume the program has been compiled into `cnt` and that all system calls succeed at runtime. How many child processes are created for the following invocations of the program? Explain how you arrived at your answer

(1) `./cnt`

(2) `./cnt 1`

(3) `./cnt 2`

(4) `./cnt 1 2 3`

b) Remove the line `exit(0)` and compile the program again. What is printed to the terminal and How many child processes are created for the following invocations of the program? Explain how you arrived at your answer.

(1) `./cnt 1`

(2) `./cnt 2`

(3) `./cnt 1 2`

(4) `./cnt 1 2 3`

Problem 11.2: *stack frames and tail recursion* (1+2 = 3 points)

As discussed in class, function calls require to allocate a stack frame on the call stack. A simple recursive function with a recursion depth n requires the allocation of n stack frames, i.e., the memory complexity grows linear with the recursion depths. In order to improve performance, compilers of high-level programming languages try to optimize the execution of recursive functions.

If a function does a function call as the last action of the function, then this function call can reuse the current stack frame. A recursive function that has this behaviour is called tail recursive. (See also Tail Recursion Explained - Computerphile on YouTube.)

Below is a definition of the function `powLin :: Integer -> Integer -> Integer` calculating the function $powLin(x, n) = x^n$.

```haskell
powLin :: Integer -> Integer -> Integer
powLin x n
     | n == 0    = 1
     | otherwise = x * powLin x (n-1)
```

a) The function `powLin` has a linear time complexity. Define a recursive function `powLog`, which has a logarithmic time complexity. You can utilize the following law:

$$pow(x, n) = \begin{cases} x^{\frac{n}{2}} \cdot x^{\frac{n}{2}} & \text{if } n \text{ is even} \\ x \cdot x^{n-1} & \text{otherwise} \end{cases}$$

b) Define a tail recursive function `powTail` with a logarithmic time complexity.

Below is a template for your solution providing some test cases.

```haskell
module Main (main) where

import Test.HUnit

powLin :: Integer -> Integer -> Integer
powLin x n
     | n == 0 = 1
     | otherwise = x * powLin x (n-1)

powLog :: Integer -> Integer -> Integer
powLog x n = undefined

powTail :: Integer -> Integer -> Integer
powTail x n = undefined

powLinTests = TestList [ map (powLin 0) [0,1,2,3,10] ~?= [1,0,0,0,0]
                       , map (powLin 2) [0,1,2,3,10] ~?= [1,2,4,8,1024]
                       , map (powLin 5) [0,1,2,3,10] ~?= [1,5,25,125,9765625]
                       ]

powLogTests = TestList [ map (powLog 0) [0,1,2,3,10] ~?= [1,0,0,0,0]
                       , map (powLog 2) [0,1,2,3,10] ~?= [1,2,4,8,1024]
                       , map (powLog 5) [0,1,2,3,10] ~?= [1,5,25,125,9765625]
                       ]

powTailTests = TestList [ map (powLog 0) [0,1,2,3,10] ~?= [1,0,0,0,0]
                        , map (powLog 2) [0,1,2,3,10] ~?= [1,2,4,8,1024]
                        , map (powLog 5) [0,1,2,3,10] ~?= [1,5,25,125,9765625]
                        ]

main = runTestTT $ TestList [powLinTests, powLogTests, powTailTests]
```

Students who prefer to write imperative code in C can solve this problem using the following C template.

```c
#include <assert.h>

```

```c
static int pow_lin(int x, int n)
{
    if (n == 0) {
        return 1;
    }
    return x * pow_lin(x, n-1);
}

static int pow_log(int x, int n)
{
    return -1;
}

static int pow_tail(int x, int n)
{
    return -1;
}

int main(void)
{
    int ns[] = { 0, 1,  2,   3,      10 };
    int t0[] = { 1, 0,  0,   0,       0 };
    int t2[] = { 1, 2,  4,   8,    1024 };
    int t5[] = { 1, 5, 25, 125, 9765625 };

    for (int i = 0; i < sizeof(ns)/sizeof(ns[0]); i++) {
        assert(pow_lin(0, ns[i])  == t0[i]);
        assert(pow_log(0, ns[i])  == t0[i]);
        assert(pow_tail(0, ns[i]) == t0[i]);
        assert(pow_lin(2, ns[i])  == t2[i]);
        assert(pow_log(2, ns[i])  == t2[i]);
        assert(pow_tail(2, ns[i]) == t2[i]);
        assert(pow_lin(5, ns[i])  == t5[i]);
        assert(pow_log(5, ns[i])  == t5[i]);
        assert(pow_tail(5, ns[i]) == t5[i]);
    }
    return 0;
}
```

Problem 11.3: *type classes (haskell)* (1+1 = 2 points)

The following Haskell module defines types for the two-dimensional shapes `Rectangle`, `Circle`, and `Triangle`.

```haskell
module Main (main) where

import Test.HUnit

data Point = Point { x :: Double, y :: Double } deriving (Show)

-- Rectangles

data Rectangle = Rectangle { p1 :: Point, p2 :: Point } deriving (Show)

-- Circles

data Circle = Circle { m :: Point, r :: Double } deriving (Show)

-- Triangles

data Triangle = Triangle { a :: Point, b :: Point, c :: Point } deriving (Show)

```

```haskell
-- Test cases

pa = Point { x =  0, y =  0 }
pb = Point { x = 10, y = 10 }
pc = Point { x =  0, y = 20 }

rect     = Rectangle { p1 = pa, p2 = pb }
circle   = Circle    { m  = pa, r  = 10 }
triangle = Triangle  { a  = pa, b  = pb, c = pc }

tests = TestList [ area rect ~?= 100.0
                 , floor (area circle) ~?= 314
                 , area triangle ~?= 100.0
                 , area (bbox rect) ~?= 100.0
                 , area (bbox circle) ~?= 400.0
                 , area (bbox triangle) ~?= 200.0
                 ]

main = runTestTT tests
```

a) Define a type class `Area` declaring a function `area`, which returns the area covered by a shape type as a `Double`. The types `Rectangle`, `Circle`, and `Triangle` shall become instances of the `Area` type class.

b) Define a type class `BoundingBox` extending the `Area` type class and declaring a function `bbox`, which returns a `Rectangle` representing the bounding box of a shape. The types `Rectangle`, `Circle`, and `Triangle` shall become instances of the `BoundingBox` type class.

Your implementation should pass the test cases.