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`

Solution:

a) Original program:

(1) The loop in `main` is never executed and hence nothing is printed and no child processes are created.

(2) The function `action` prints `(1,1)` and then a child process is created that calls `action`. The function call of the child process prints `(1,0)` and then the child process exits.
(3) The function `action` prints `(2,2)` and then a child process is created that calls `action`. The child's call of action prints `(2,1)` and then another child is created that calls `action`. The second child will print `(2,0)` and no create any further child processes.
(4) Following the previous logic, the first argument creates 1 child process, the second argument 2 child processes, and the third argument 3 child processes. Hence, a total of 6 processes are created.

b) After the removal of `exit(0)`:

(1) Like before, the first call of `action` will print `(1,1)` and a child process is created calling `action` and printing `(1,0)`. The child process does not exit and hence it will return to the `main` function. Since the loop in the `main` function is at the end, the child process will eventually exit from `main` (as does the parent process).
(2) Like before, two child processes are created. Both return to `main` but since the loop in the `main` function is at the end, the two child processes will eventually exit from `main` (as does the parent process).
(3) The first call of `action` from —main— leads to a child process and both the parent and the child process return from `action`. Since there is one more iteration of the the loop in the `main` function, both processes will call action again, each call causing the creation of two child processes. Hence, in total, 5 child processes are created.
(4) Like before, the first argument leads to the creation of one child process and both the parent and the child process continue with the execution of the loop in the `main` function. The second argument then leads to 4 additional child processes. Hence, the last argument is processed by 6 processes and each of the 6 calls to `action` lead to 3 additional child processes. Hence, a total of 1+4+18 = 23 child processes are created.

Marking:

a)
- *0.2pt for a correct answer and explanation for (1)*
- *0.4pt for a correct answer and explanation for (2) and (3)*
- *1pt for a correct answer and explanation for (4)*

b)
- *0.5pt for a correct answer and explanation for (1) and (2)*
- *1pt for a correct answer and explanation for (3) and (4)*

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 $powLin(x, n) = x^n$.

```
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>

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;
}
```

Solution:

A solution for terse Haskell programmers:

```
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
    | n == 0     = 1
    | even n     = p * p
    | otherwise = x * powLog x (n-1)
    where p = powLog x (n `div` 2)

powTail :: Integer -> Integer -> Integer
powTail x n = go x n 1
    where go x 0 a = a
          go x n a
            | even n     = go (x * x) (n `div` 2) a
            | otherwise = go x (n-1) (x * a)

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

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

A solution for verbose C programmers:

```c
#include <assert.h>

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)
{
    if (n == 0) {
        return 1;
    }
    if (n % 2) {
        return x * pow_log(x, n-1);
    } else {
        int p = pow_log(x, n / 2);
        return p * p;
    }
}

static int pow_tail_go(int x, int n, int acc)
{
    if (n == 0) {
        return acc;
    }
    if (n % 2) {
        return pow_tail_go(x, n-1, x * acc);
    } else {
        return pow_tail_go(x * x, n / 2, acc);
    }
}

static int pow_tail(int x, int n)
{
    return pow_tail_go(x, n, 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;
}
```

Marking:

a) - *1pt for a logarithmic time recursive function*

b) - *2pt for a logarithmic time tail recursive function*

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)

-- 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.

Solution:

```haskell
module Main (main) where

import Test.HUnit

class Area a where
  area :: a -> Double

class Area a => BoundingBox a where
  bbox :: a -> Rectangle

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

-- Rectangles

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

instance Area Rectangle where
  area r = dx * dy
    where dx = if x1 < x2 then x2 - x1 else x1 -x2
          dy = if y1 < y2 then y2 - y1 else y1 -y2
          x1 = x (p1 r)
          y1 = y (p1 r)
          x2 = x (p2 r)
          y2 = y (p2 r)

instance BoundingBox Rectangle where
  bbox r = r

-- Circles

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

instance Area Circle where
  area c = pi * (r c) * (r c)

instance BoundingBox Circle where
  bbox c = Rectangle { p1 = Point { x = mx - rr, y = my - rr }
                     , p2 = Point { x = mx + rr, y = my + rr } }
           where mx = x (m c)
                 my = y (m c)
                 rr = r c

-- Triangles

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

instance Area Triangle where
  area t = 0.5 * (abs xa*yb - xa*yc + xb*yc - xb*ya + xc*ya - xc*yb)
    where xa = x (a t)
          xb = x (b t)
          xc = x (c t)
          ya = y (a t)
          yb = y (b t)
          yc = y (c t)

instance BoundingBox Triangle where
  bbox t = Rectangle { p1 = Point { x = minimum [xa, xb, xc],
                                    y = minimum [xa, xb, xc] }
                     , p2 = Point { x = maximum [xa, xb, xc],
                                    y = maximum [ya, yb, yc] } }
```

```haskell
    where xa = x (a t)
          xb = x (b t)
          xc = x (c t)
          ya = y (a t)
          yb = y (b t)
          yc = y (c t)

-- 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
```

Marking:

a) - *0.5pt for a correct definition of the* `Area` *type class*
   - *0.5pt for making the shape classes instances of* `Area`

b) - *0.5pt for a correct definition of the* `BoundingBox` *type class*
   - *0.5pt for making the shape classes instances of* `BoundingBox`