Intro to Computer Science Assignment 10

Yiping Deng

2017-11-23

1.

a) $\circ$ is associatetive, e is neutral element $\implies (x \circ y) \circ z = x \circ (y \circ z)$
$x \circ e = x$
using this property, we can prove the inductive basis

```
foldl op e [] = e
foldr op e [] = e
foldl op e [] = foldr op e []
```

inductive step:

```
foldl op e (x:xs) = (op e x) `op` (foldl op e xs) = x `op` (foldl op e xs)
foldr op e (x:xs) = x `op` (foldr op e xs)
-- by inductive hypothesis
foldl op e xs = foldr op e xs
-- which implies
foldl op e (x:xs) = x `op` (foldl op e xs) = x `op` (foldr op e xs) = foldr op e (x:xs)
```

b) proof:
using induction, first we need to prove the basis:

```
foldr op1 e [] = e = foldl op2 e []
```

inductive step:

```
-- assumption
foldr op1 e xs = foldl op2 e xs
-- proof
foldl op2 e (x:xs) = foldl op2 (e `op2` x) xs
= foldl op2 (x `op1` e) xs
-- using the first rule
= x `op1` (foldl op2 e xs)
```

```haskell
-- by induction hypothesis
= x `op1` (foldr op1 e xs)
-- put it inside of foldr
= foldr op1 e (x:xs)
```

c) proof:
inductive basis:

```haskell
foldr op a [] = a = foldl op` a [] = foldl op` a (reverse [])
```

inductive step:

```haskell
foldl op` a (reverse (x:xs)) =
-- since it is reversed
= (foldl op` a (reverse xs)) `op`` x
-- by inductive hypothesis
= (foldr op a xs) `op`` x
-- by using the rule
= x `op` (foldr op a xs)
-- by moving inside
= foldr op a (x:xs)
```

2.

a) For invocation "./happy-fork", there is no extra argument $\Longrightarrow$ $argc ==$ 1 (the first argument refers to itself) $\Longrightarrow$ the main process never goes into the for loop $\Longrightarrow$ there will be 0 child process(0 fork invocation)

For invocation "./happy-fork a", there is 1 extra argument $\Longrightarrow$ $argc == 2$ $\Longrightarrow$ the main process will be in the loop once $\Longrightarrow$ the direct child of main will never call the loop again, and the direct child will call fork() one more time $\Longrightarrow$ there will be 2 child processes

For invocation "./happy-fork a b", there is 2 extra argument $\Longrightarrow$ $argc == 3$. The first two fork will have the exact situation as the main process in $argc == 2$, which will have 3 process(including the main). The main will also behaves like program $argc == 2$. In total, $(2+1)*3 = 9$ process, which means 8 subprocess.

We can easily conclude that every time the program will split into 3 process that behaves like the argc is $argc - 1$.
Hence, number of subprocess $= 3^{argc-1} - 1$
Therefore, "./happy-fork a b c" will have 26, and "./happy-fork a b c d" will have 80.

This conclusion matched the subprocess counter I created.

```c
#include <stdio.h>
#include <unistd.h>
int main(int argc, char** argv){
    int fd[2];
    int depth = 0; /* keep track of number of generations from original */
    int i;
    pipe(fd);
    for(; argc > 1; argc--){
        if(0 == fork()){
            (void) fork();
            write(fd[1], &i, 1);
            depth += 1;
        }
    }
    close(fd[1]);  /* close the pipe so that it will not wait forever */
    if( depth == 0 ) { /* original process */
      i=0;
      while(read(fd[0],&depth,1) != 0)
        i += 1;
      printf( "%d total processes spawned\n", i);
    }
    return 0;
}
```

This program uses pipes to keep track of number of subprocess. This matches my conclusion perfectly.

2) This is a program that does the same thing as the previous, short C program, without stdlib, only system call.

```asm
.text
    .global _start

_start:
    mov    (%rsp), %r8 ## use r8  to store argc
.LOOP_START:
    cmp    $2, %r8 ## compare with 2, if smaller, skip loop
    jl     .LOOP_END ## end the loop if small than 2
    mov    $57, %rax ## system call number
    syscall          ## make the system call
    test   %rax, %rax
    jne    .IF_END ## if it is not 0, jump to the end of if
    mov    $57, %rax ## take the system call fork number
    syscall         ## make the system call, again
.IF_END:
    dec    %r8        ## minus 1 to the rcx(the argc)
```

```
    cmp    $1, %r8    ## compare r8 and 1
    jg    .LOOP_START  ## jump back and start the loop if greater
.LOOP_END:

    ## use different approach to end the program

    movl    $len,%edx       ##  third argument: message length.
    movl    $msg,%ecx       ## second argument: pointer to message to write.
    movl    $1,%ebx                ## first argument: file handle (stdout).
    movl    $4,%eax                ## system call number (sys_write).
    int     $0x80           ## call kernel.

    ## and exit.

    movl    $0,%ebx         ## first argument: exit code.
    movl    $1,%eax         ## system call number (sys_exit).
    int     $0x80           ## call kernel.

.data

msg:
        .ascii  "x\n"      # the string to print.
        len = . - msg                  # length of the string.
```