

CSC3150 Project 2 report

119010341 WU Yifan

Program Design

1. Multi-Thread Programming

In total we use 10 threads to implement the game (1 thread controlling the frog's movements, and the other 9 controlling the logs' movements).

We use `pthread_create()` to create threads, `pthread_join()` to join the tails of different threads, and `pthread_mutex_init()` to generate a mutex for the parallel programming.

When creating the log threads, we pass the `long` type `log_index` to the thread function `logs_move()` (`log_index` equals to 0, 1, 2, 3, 4, ...8).

2. Constructing The Game Map

In the first part of the assignment, we are required to display the game in the terminal.

We should continuously print out the map of game in the terminal.

So the best variable type for the map of the game is a **character array**.

```
char map[ROW][COLUMN];
```

Then in the `map` we need different ASCII for the frog, the logs and the river bank.

```
map[...][...] = '0'; // the frog
map[...][...] = '='; // the logs
map[...][...] = '|'; // the river bank
```

So how do we insert these three elements into the map specifically?

1. For the frog, we use the data structure `Node frog`, which consist of `frog.x` and `frog.y` as its coordinates.

```
frog = Node( ROW, (COLUMN-1) / 2 );
```

2. For the logs, we use an integer called `head_index` to denote the starting point of a log, when the length of the log is 15, we can simply use the subsequent coordinates (such as `head_index+1`, `head_index+2`, ...`head_index+15`) to find the coordinates of the whole log. Each `head_index` is firstly random generated in the map.

```
head_index = rand()%48;
```

3. For the river bank, we just need to input them into the map when initializing the game map (put '|' on the upper bound and the lower bound of the map).

3. Game status

To distinguish different game status, we introduced a set of global variables

1. `int isQuit`

When `isQuit == 1`, either the user has exited the game, or the user has won or lost the game.

2. `int isOnLog`

When the frog jumps onto a log, `isOnLog` will change to the `log_index` of the respective log. Otherwise `isOnLog` will remain 0.

4. The Movement of The Frog

This part is included in the thread function `void *frog_move(void *a){}.`

Using a `while` loop together with the `kbhit()` and `getchar()`, we can keep monitoring the keyboard input from the user.

```
while (!isQuit){
    if( kbhit() ){
        char dir = getchar() ; // dir means "direction"
        ...
    }
}
```

Then we can map different cases of the value of `dir` to different operations on the frog. For example, when `dir = 'w'`, we deduct 1 from `frog.x`, which means the frog moves up by 1 block.

Do notice that in this program we use 'x' for the vertical coordinate and 'y' for the horizontal coordinate of the frog

However, before we update `frog.x` and `frog.y`, we first need to memorize last place where the frog has been, so that it is easy to recover the character in that place to what it used to be.

For example,

```
map[frog.x][frog.y] = last_step; // recover last step
last_step = map[frog.x][frog.y]; // memorize the new one
if(dir == 'w' && dir == 'W') // update frog coordinates
    if(frog.y < 48){ // cannot go beyond the map
        frog.y += 1;
    }
map[frog.x][frog.y] = '0'; // update the map
```

Notice that when the frog is on the starting river bank, we cannot let it to go beyond the game map, so we need to add the conditional statement.

After the frog makes each movement, we need to immediately judge the current game status,

Whether the frog has jumped into the river?

Whether the frog has reached the other side?

So we use a function called `JudgeStatus()` to complete this task.

Based on the character value at the new coordinate, we can determine the new game status.

1. If `map[frog.x][frog.y] == '|'`, the frog is on the river bank, and we can tell from `frog.y` which river bank the frog is on.
2. If `map[frog.x][frog.y] == '='`, the frog has jumped onto a log.
3. If `map[frog.x][frog.y] == ' '`, the frog has jumped into the river.

5. The Movement of The Logs

This part is included in the function `void *logs_move(void *t)`

By type casting, `log_index` equals to `t`

Since logs need to move staggerly, we can introduce a integer called `direction` to denote the moving direction of each log. When `log_index` is an odd number, `direction` is 1. When `log_index` is an even number, direction is -1.

```
direction = (log_index % 2)*2 -1 // odd row moves right
```

So we can add `direction` to the `head_index` to make the logs move.

We use the **modular arithmetic** to keep the logs cycling in the map.

For example,

```
if(direction==1){
    head_index = (head_index + direction)%(COLUMN-1); // moves right
    // input the log into the map
    for(int i=0; i<15; i++){
        if((head_index+i)%(COLUMN-1)==frog.y && log_index==frog.x){
            // include the frog
            map[log_index][(head_index+i)%(COLUMN-1)] = '0';
        }else{
            map[log_index][(head_index+i)%(COLUMN-1)] = '=';
        }
    }
}
```

Also, remember to include the frog in the corresponding log if the frog is on a log.

So that `head_index` as well as the subsequent coordinates will change from 48 to 1 if the log is moving rightward

After the frog jumps onto a log, it will move along with the log. So we use conditional statement to implement this requirement.

When the frog is on a specific log, the game status variable `isOnLog` equals to the `log_index` of that log. We can compare `isOnLog` with `log_index`.

```
if(last_step=='=' && log_index == (long)isOnLog){
    frog.y += direction;
}
```

Also, remember to check if the frog floats out of the map along with the log. If this happens, the user lose the game.

Running Environment

OS, Kernel, and g++

```
wyf@ubuntu:~/Desktop/CSC3150/assignment_2/bonus$ cat /etc/issue
Ubuntu 16.04.5 LTS \n \l

wyf@ubuntu:~/Desktop/CSC3150/assignment_2/bonus$ g++ --version
g++ (Ubuntu 5.4.0-6ubuntu1~16.04.12) 5.4.0 20160609
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

wyf@ubuntu:~/Desktop/CSC3150/assignment_2/bonus$ uname -r
4.15.0-142-generic
```

```
The Linux version is 16.04.5
The Linux kernel version is 4.15.14
The g++ version is 5.4.0
```

Bonus Task: OpenGL (glut package)

To set up running environment for OpenGL, please follow the guidelines below

1. Install OpenGL Library

```
$ sudo apt-get install libgl1-mesa-dev
```

2. Install OpenGL Utilities

```
$ sudo apt-get install libglu1-mesa-dev
```

3. Install OpenGL Utility Toolkit

```
$ sudo apt-get install freeglut3-dev
```

Program Execution

Non-Bonus Part

cd into the `source` folder

```
$ cd source
```

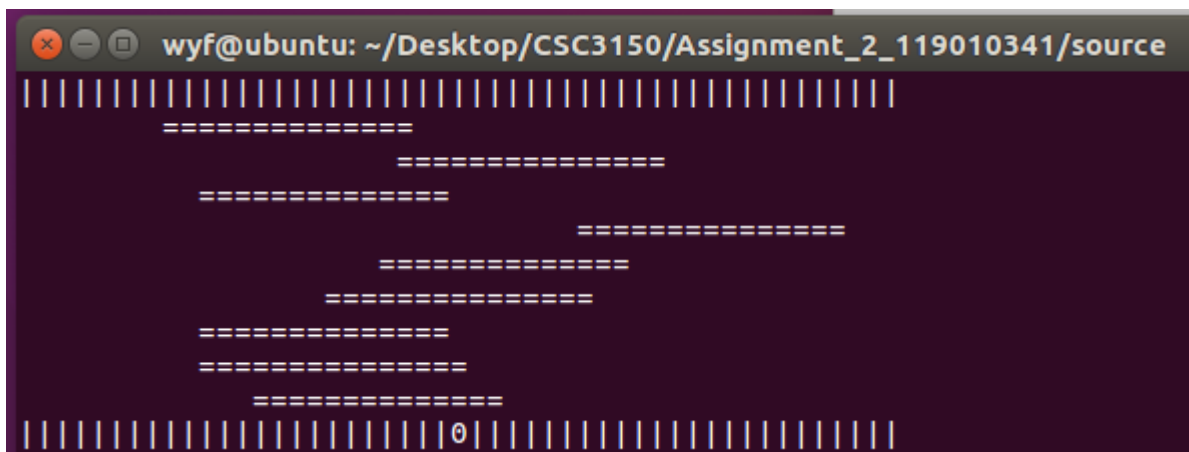
compile the program

```
$ g++ hw2.cpp -lpthread
```

run the program

```
$ ./a.out
```

Sample Output



```
wyf@ubuntu: ~/Desktop/CSC3150/Assignment_2_119010341/source
|||||
=====
          =====
        =====
      =====
    =====
  =====
=====
|||0|||
```

Bonus Part

cd into the bonus folder

```
$ cd bonus
```

compile the program

```
$ g++ hw2.cpp -lGL -lGLU -lglut -lpthread
```

run the program

```
./a.out
```

Sample output

The sample output is placed in the bonus directory as a [video named "demo.mp4"](#)

What I have learned

Multi-thread Programming

This assignment gives me a chance to feel the convenience of multi-thread programming. I can see that it is quite useful in game designing. When you need to constantly monitor the keyboard input, and use while loop to move the logs, multi-programming can make your threads run simultaneously and parallelly.

Besides, `mutex` is very effective when we are doing multi-thread programming since some of the threads may need to access same blocks of data in the memory, and it can result in data hazard if we don't lock the threads. This deepens my understanding that threads are in the same process, and they share the same block of memory.

Also, by join the tails of the threads together, we can prevent the main thread from terminating the other threads.

However, I do realize that this assignment is not that perfect as a multi-thread orientation programming project.

1. It can be done with a single thread.
2. In the program design, there are many global variables which may be accessed by most threads. In this case, no matter thread is functioning, it probably needs to use `mutex` to lock up the threads. As a result, all threads are almost running concurrently instead of simultaneously, which means that there isn't much advantage using multi-thread programming.

Graphical Output

There are many differences between terminal output and graphical output.

Terminal output can directly print the characters onto the terminal, which is fast. Graphical output, on the other hand, it need to draw the graphs, which is quite time-consuming. So the graphs must first be pushed into a buffer, and after that be rendered onto the window.

It is quite important to think through when and how you should render these graphs. I have encountered a problem where the graphical window just stuck and nothing came out. It turned out that it was because I rendered the graphs so frequently that the computer had no time push the items into its buffer. So I had changed the display mode from "Single-Buffered" to "Double-Buffered" and adjusted the display frequency.