CSC3150 Assignment2 Report

Introduction

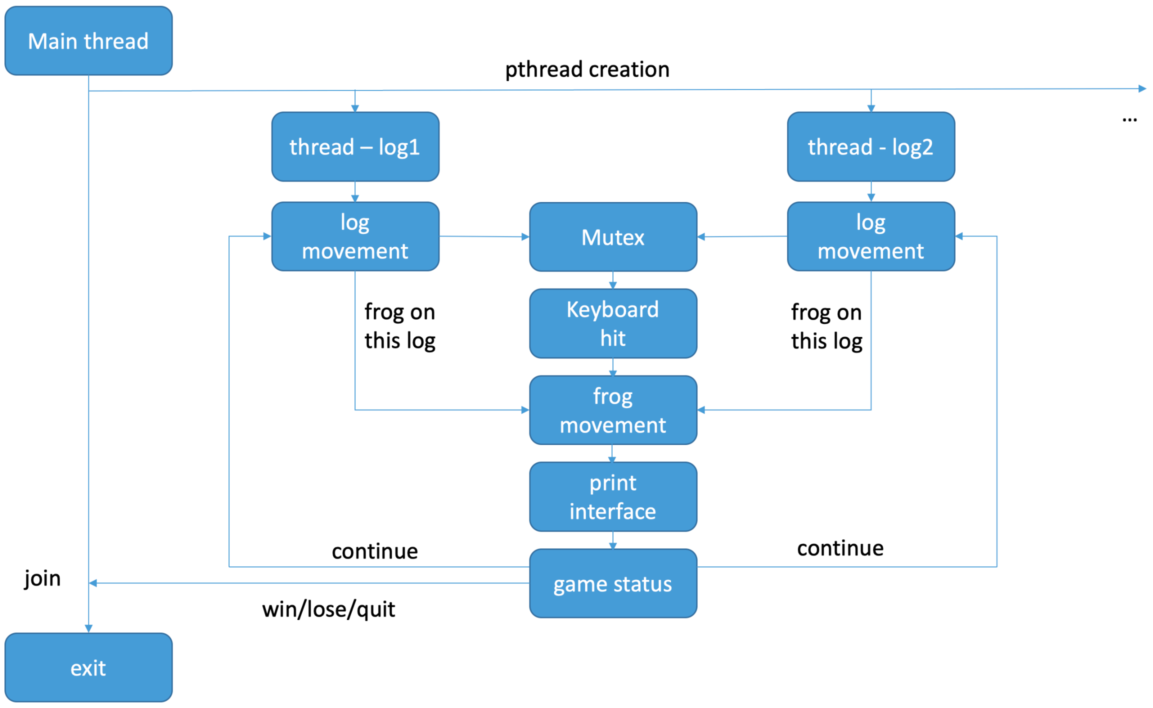
In this assignment, a game "frog cross river" is implemented by using pthread. Also, the length of log could be randomly generated, which is one of the bonus requirements. The source code is in the file 'hw2.cpp', which is attached with this report.

Program design

To implement the game, the program could be roughly divided into following parts:

- Display of the game interface.
- Creation and movement of logs.
- Control of the frog movement.
- Judgment of the game state.

After every part is implemented, they should be integrated as a game that ready for users to play. The following graph shows the main logic that combines these parts.



According to this logic graph, when the game begins, the program would do the following things step by step and these step will be elaborated with actual code:

1. First, preparation for global variables:

```
#define ROW 10
#define COLUMN 50
```

```
pthread_mutex_t mutex;

int state = 0;  // on-going:0 ; win:1 ; lose:2 ; quit:3
char buf = '|';      // used for frog movement
char map[ROW+10][COLUMN];

struct Node{
    int x , y;
    Node( int _x , int _y ) : x( _x ) , y( _y ) {};
    Node(){} ;
} frog ;
```

where 'map' is the basic place we playing on and its size depends on the value of ROW and COLUMN.

The variable 'frog' with struct Node represents the position of flog in the map. Since there is only one flog during the whole game, every thread later is supposed to share these single variable, and so as the variable 'state', which is initialized as 0 to represent the on-going state of the game. The rest variables will be explained later.

2. After the global variables have been prepared, in the main() function,
    1. Initialize the map of the game:

    ```
    // Initialize the river map and frog's starting position
    memset( map , 0, sizeof( map ) ) ;
    int i , j ;
    for( i = 1; i < ROW; ++i ){
        for( j = 0; j < COLUMN - 1; ++j )
            map[i][j] = ' ' ;
    }
    for( j = 0; j < COLUMN - 1; ++j )
          map[ROW][j] = map[0][j] = '|' ;
    frog = Node( ROW, (COLUMN-1) / 2 ) ;
    map[frog.x][frog.y] = '0' ;
    ```

    2. Create pthreads for every log (row) in the river:

    ```
    /*  Create pthreads for wood move and frog control.  */
    pthread_t threads[ROW - 1];
    long tids[ROW - 1]; // used for id allocation
    for (i = 0; i < ROW - 1; ++ i){
          tids[i] = i;
    }
    int rc;
    srand(time(0)); // used to avoid pseudo-random
    pthread_mutex_init(&mutex, NULL); // initialize the mutex
    printf("\033[H\033[2J");
    ```

```
for (i = 0; i < ROW - 1; ++ i){
    rc = pthread_create(&threads[i], NULL, logs_move, (void*)tids[i]);
    if (rc) {
        printf("ERROR: return code from pthread_create() is %d", rc);
        exit(1);
    }
}
```

Here, number from 0 to (ROW-2) are used as the id for each log (thread).

3. Wait until the threads exit:

```
pthread_exit(NULL);
pthread_mutex_destroy(&mutex);
```

3. After the pthreads have been created, in function logs_move():

    1. Randomly generate the start point and the length of each log:

    ```
    int LOG_SIZE;
    LOG_SIZE = (COLUMN / 8) + (rand() % (COLUMN / 3));
    int lx = rand() % (COLUMN - 1); // the left-end position of the log
    ```

    Here, the length of the log can be from 1/8 to 3/8 width of the river long, which meets one of the requirements of bonus problems. ⬅️ Also the start position of the log is randomly generated, too.

    2. Move the logs:

    ```
    frog_on_log = 0;
    // for the log with even id, move towards right
    if (tid % 2 == 0){
        if (frog.x == tid + 1) frog.y += 1; // move frog if frog is at
    this row
        map[tid + 1][lx] = ' '; // remove left-end '='
        for (i = lx + 1; i < lx + 1 + LOG_SIZE; ++ i){
            map[tid + 1][i % (COLUMN-1)] = '=';
            if (frog.x == tid + 1 && i % (COLUMN-1) == frog.y) frog_on_log
    = 1;
        }
        lx = (lx + 1) % (COLUMN - 1);
    }
    // for the log with odd id, move towards right
    else{
        if (frog.x == tid + 1) frog.y -= 1; // move frog if frog is at
    this row
        map[tid + 1][(lx + LOG_SIZE) % (COLUMN - 1)] = ' '; // remove
    right-end '='
        for (i = lx; i < lx + LOG_SIZE; ++ i){
            map[tid + 1][i % (COLUMN - 1)] = '=';
    ```

```
        if (frog.x == tid + 1 && i % (COLUMN-1) == frog.y) frog_on_log = 1;
    }
    lx = (lx + COLUMN - 2) % (COLUMN - 1);
}
if (frog.x != tid + 1) frog_on_log = 1;
map[frog.x][frog.y] = '0' ;
```

The movement of the log is implemented by removing the '=' on one side and adding '=' on the other side. And it uses the left-end position of the log is recorded for later movement. Moreover, for the log with even id, move it towards right; for the log with odd id, move it towards left.

Also, as the previous logic graph shows, the log movement also control the movement of the frog, since the frog will move with the log that it stands on. The boolean variable 'frog_on_log' is used to judge whether the frog is on one of the logs, when the frog is not on the bank, by checking all the logs. This 'frog_on_log' is used for later game state judgment.

3. (**Where did you place the mutex lock and why?**) ⬅️

   After the log movement, each log trying to access the mutex, as you can see in the previous logic graph. The mutex here is important and necessary since the operations inside the mutex could only be execute once at every single time slot.

   ```
   pthread_mutex_lock(&mutex);
   ```

   For keyboard hit detection and the frog movement. It's easy to understand that, if every thread detects a single keyboard hit and moves the frog at the same time, the final displacement of the frog would be much larger instead 1 unit.

   For interface printing, if every thread print the interface concurrently, the output will be messed up since the threads print their own result in turns.

   The following is the detailed steps:

   1. Change the frog position when the related keyboard hit occurs. A buffer 'buf' is used to record the symbol the frog step to so that the symbol is kept the same as before after the frog passed by:

      ```
      /*  Check keyboard hits, to change frog's position or quit the game. */
      if (kbhit()){
          char dir = getchar();
          map[frog.x][frog.y] = buf;  // make the place get its previous symbol
          if ((dir == 'w' || dir == 'W') && frog.x > 0) // move up
              frog.x -= 1;
          if ((dir == 's' || dir == 'S') && frog.x < ROW) // move down
              frog.x += 1;
          if ((dir == 'a' || dir == 'A') && frog.y > 0) // move left
      ```

```
        frog.y -= 1;
    if ((dir == 'd' || dir == 'D') && frog.y < COLUMN - 2)  //
move right
        frog.y += 1;
    if (dir == 'q' || dir == 'Q') // quit the game
        state = 3;  // game state: quit
    // record the symbol on the place that the frog is going to
step on
    buf = map[frog.x][frog.y];
    map[frog.x][frog.y] = '0' ;
}
```

2. Check the game state:

   ```
   // win if frog arrive the upper bank
   if (frog.x == 0) state = 1;
   // lose if frog reach the left side or right side or if frog steps
   into the river
   if (frog.y == -1 || frog.y == COLUMN - 1 || !frog_on_log) state =
    2;
   ```

   The check of state 3 (quit action) is inside the keyboard hit detection in step 1.

3. Print the real-time interface on the screen:

   ```
   printf("\033[H\033[2J");  // clear the screen
   for(i = 0; i <= ROW; ++i)
       puts( map[i] );
   ```

4. Display the result if the game is not on on-going state:

   ```
   if (state == 1) printf("\033[H\033[2JYou win the game!!\n");
   if (state == 2) printf("\033[H\033[2JYou lose the game!!\n");
   if (state == 3) printf("\033[H\033[2JYou exit the game.\n");
   ```

Finally we could unlock the mutex:

```
pthread_mutex_unlock(&mutex);
```

As you could see in the logic graph, after these three operations inside the function logs_move(), it will goes back to log movement again, until the game is over by the change of status. Therefore, all the operations of step 2 and 3 were in a while loop:

```
while (state == 0) {
    usleep(100000);   // the value is proportional to the speed of
movements
    ...
}
```

This makes the logs keep moving and the interface keep being refreshed, until game is over, then we could exit the pthread, and back to the main thread, and the running of the program is finished.

```
pthread_exit(NULL);
```

Problems I met

Problem

One of the problems I met is that the output of the interface was messed up during the game.

Solution

Reduce the number of threads to 2, and then find out that it is because the mutex was not used so that two thread print concurrently.

Steps to run the program

1. In the directory that contains the file '*hw2.cpp*', type the following command and enter to compile it:

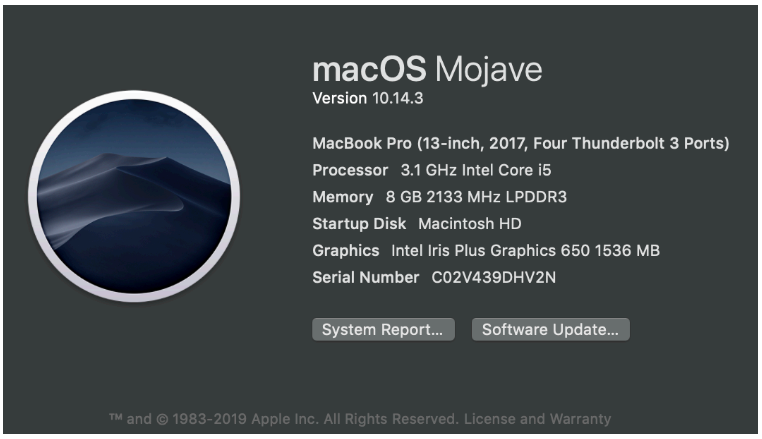
```
g++ hw2.cpp -lpthread
```

2. In the same directory, type the following command and enter to run it:

```
./a.out
```

Program output

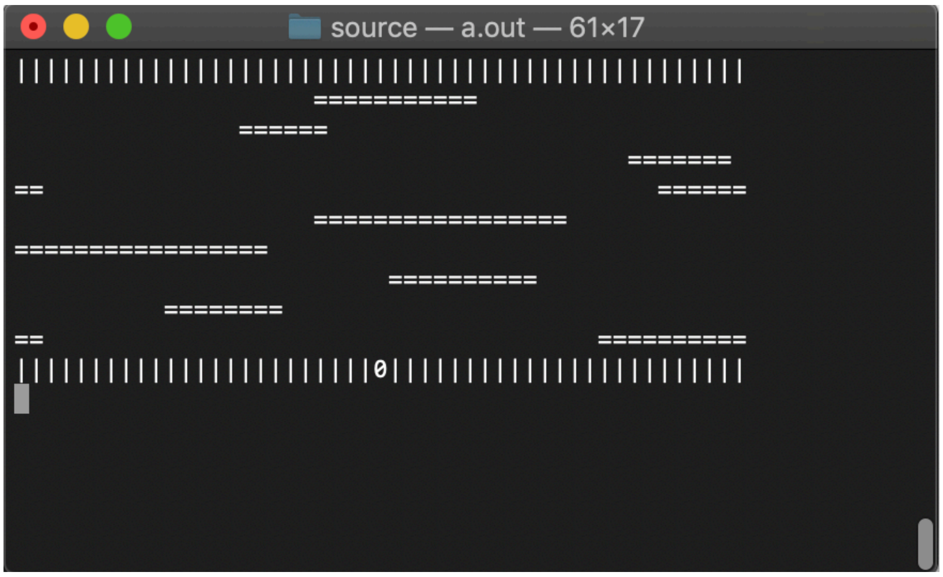
Running environment


macOS Mojave
Version 10.14.3

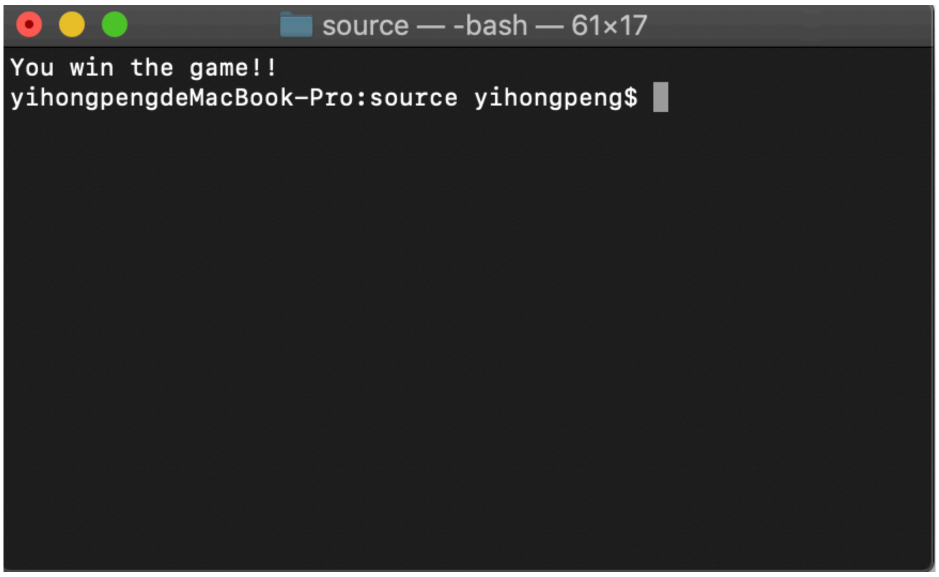
MacBook Pro (13-inch, 2017, Four Thunderbolt 3 Ports)
Processor 3.1 GHz Intel Core i5
Memory 8 GB 2133 MHz LPDDR3
Startup Disk Macintosh HD
Graphics Intel Iris Plus Graphics 650 1536 MB
Serial Number C02V439DHV2N

™ and © 1983-2019 Apple Inc. All Rights Reserved. License and Warranty

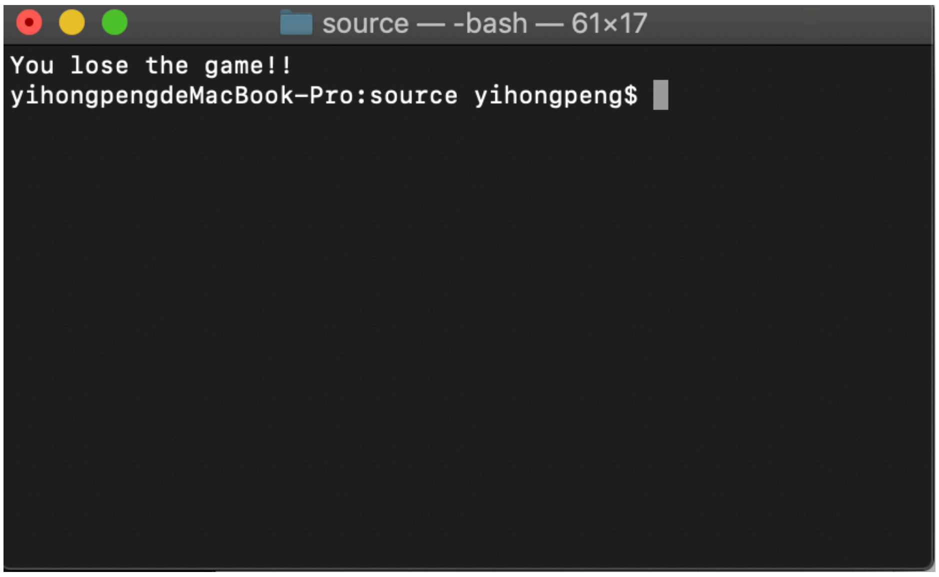
Sample output



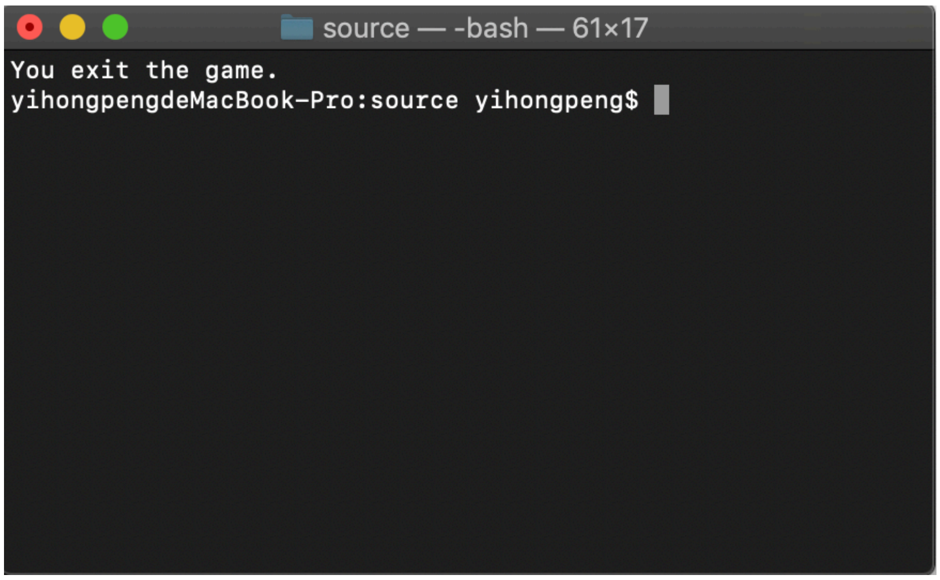
```
|||||||||||||||||||||||||||||||||||||||||||||||||
                    ===========
               ======
                                         =======
==                                         ======
                    =================
=================
                         ==========
          ========
==                                     ==========
||||||||||||||||||||||||0||||||||||||||||||||||||
```



```
You win the game!!
yihongpengdeMacBook-Pro:source yihongpeng$
```



```
You lose the game!!
yihongpengdeMacBook-Pro:source yihongpeng$
```



```
You exit the game.
yihongpengdeMacBook-Pro:source yihongpeng$ 
```

What I learned

- What a multithread program could do
  - similar things following the same rule
- How to design a multithread program
  - parallel thinking
  - deeper understanding on the use of mutex lock
  - deeper understanding on the operating system
- How to use pthread to implement a program
  - multithread programming skill