CSC3150 Assignment2 Report

Name: Hu Yiyan

Student id: 119010104

Date: October 23th, 2021

Design

Threading

- Overall Design - pthread

> The game contains three parts:
>
> (1) moving the log in the river -> log move
>
> (2) controlling the frog to jump across the logs -> frog control
>
> (3) updating game interface and checking game status -> game control
>
> To make the three processes run concurrently, we can use threads, for thread is:
>
> > "an independent stream of instructions that can be scheduled to run as such by the operating system"
>
> We can create three pthreads to make the three parts of the program run concurrently. I create three pthreads and each of them perform a part of the game:

```
void *thread_logs_move( void *t );  //Game part1: Log move
void *thread_move_control(void *t); //Game part2: frog control
void *thread_game_control(void *t); //Game part3: game control
```

- Thread Creation and Synchronization - pthread_create & pthread_join

> To use the pthread, we need to create pthreads. We call pthread_create() to create a thread. The following is an example

```
// create pthreads
// the program will generate an error if the creation fails
if (pthread_create(&th[ROW], NULL, thread_game_control, NULL) != 0){
    perror("Failed to create thread");
}
// note: &th[row]: pass the thread id, thread_game_control: the
attribute function of the thread
```

> To make the threads run concurrently, we need to use the pthread_join() to synchronize threads. Below is an example

```
// join pthread with thread_id = th[ROW]
// the program will generate an error if the joining fails
if (pthread_join(th[ROW], NULL) != 0){
    perror("Fail to join thread");
}
```

We also need to remember to exit the thread when the thread completes, using the pthread_exit().

- Thread Privacy and Security - Mutex lock

> The threads work concurrently, when multiple threads try to access or modify same data, there can be security issues. To avoid the shared data security problems, we can use pthread_mutex to set privacy and security flags.
>
> In my program, there are three threads. I create three thread_mutex to protect shared data in the program. Each time the program deal with shared data, only one mutex will be unlocked.

```
pthread_mutex_t mutexMap;  // mutex for the log move thread
pthread_mutex_t mutexFrog; // mutex for the forg control thread
pthread_mutex_t mutexGame; // mutex for the game control thread
```

> A mutex variable acts like a "lock" protecting access to a shared data resource, we can pthread_mutex_lock() to protect the shared data. Below is an example.

```
pthread_mutex_lock(&mutexGame); // lock mutex
// perform on shared data
pthread_mutex_unlock(&mutexGame); // unlock mutex
```

- The Log move thread - thread_logs_move()

> One pthread controls the log movement. The attribute function of the pthread is the thread_logs_move().

```
void *thread_logs_move( void *t ){...}
// we can read the thread id to distinguish the nine logs on the river
and control their movements
```

Keyboard Input Control

read input from keyboard: kbhit()

- Read Input from Keyboard - kbhit()

> To make the player control the frog, we need to read the player's input and perform operations on frog accordingly. Reading player inputs is implemented by the kbhit() and getchar() functions.
>
> We first call kbhit() to check if the player hit the keyboard. Then we call getchar() to get the key hit by player and perform operations on frog.

- Controlling the frog - thread_move_control()

> We use one pthread to control the movement of the frog. The attribute function of the pthread is thread_move_control(). The control will only involve the frog postion (i.e. frog.x and frog.y), it will not change the map.
>
> ```
> void *thread_move_control(void *t){...}
> // in the function,
> // we first read the key, then perform operations on the frog
> ```

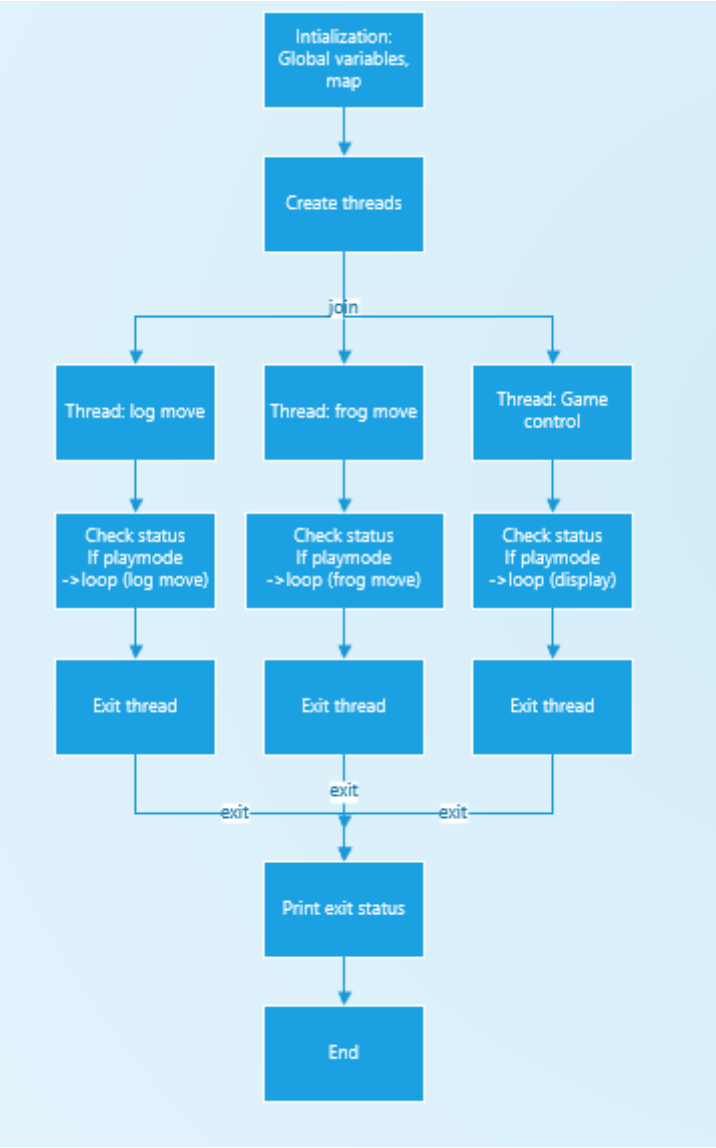
Game Control

- Control the Game - thread_game_control()

> We use one thread to control the game. We control the game by: (1) refreshing the game status, (2) display the game in the terminal.
>
> The logic of the attribute of the game control pthread is as follows
>
> ```
> void *thread_game_control(void *t){
>     //while checkGamestatus != endGame
>     //  displayGame
>     //Exit the thread
> }
> ```
>
> When the thread checks the game status, it will change the game flags. The pthread will exit if the LOSE or EXIT or WIN flag is set.

Program Logic

- Flow chart - logic of the designed program



Code Design

- Encapsulate Codes

> I try to make the code in the thread attribute functions as clear as possible. I encapsulate the codes inside the attribute functions into subroutines. Below is an example. In the game control thread attribute function, I encapsulate the status checking codes and terminal displaying codes in other subroutines.

```
// subroutines:
// control_GameStatusPlaying(): check status
// changeMap_PrintRiverMap();: terminal display
void *thread_game_control(void *t){
    while (control_GameStatusPlaying()){
        changeMap_PrintRiverMap();
        usleep(LAG);
    }
    pthread_exit(NULL);
}
```

- Categorize Functions

> There are quit a few functions in the code file. I try to organize the functions in different categories. I rename the functions according to their categories.

```
void control_ChangeGameStatus(int status);
// function in the control category
void changeMap_frog(int x, int y)
// function in the change map category
void *thread_logs_move( void *t )
// function in the thread category
```

Note: I put all the initializations in the main().

Environment to Run the Program

Linux Version

Check the Linux version:

```
timothy@ubuntu:~$ cat /etc/issue
Ubuntu 16.04.7 LTS \n \l
```

GCC Version

Check the GCC version

```
timothy@ubuntu:~$ gcc --version
gcc (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.
```

Execute the Program

To execute the program, there are 2 ways

- Way 1:

  Compile the code via command: gcc hw2.cpp -lpthread

  Execute the program via command: ./a.out

```
timothy@ubuntu:~/Projects/Ass2/CSC3150_project_2/source$ gcc hw2.cpp -lpthread
timothy@ubuntu:~/Projects/Ass2/CSC3150_project_2/source$ ./a.out
```

- way 2:

Type the command: make

I have written a Makefile so that you can compile and execute by simply typing "make".

```
timothy@ubuntu:~/Projects/Ass2/CSC3150_project_2/source$ make
```

Output

- Gaming Display

  The game is displayed on the terminal.

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

- Print game ending status

  There are 4 game status: PLAY, WIN, LOSE and EXIT, when the game stops. The terminal will show the ending status. The ending status can only be one of the WIN, LOSE and EXIT.

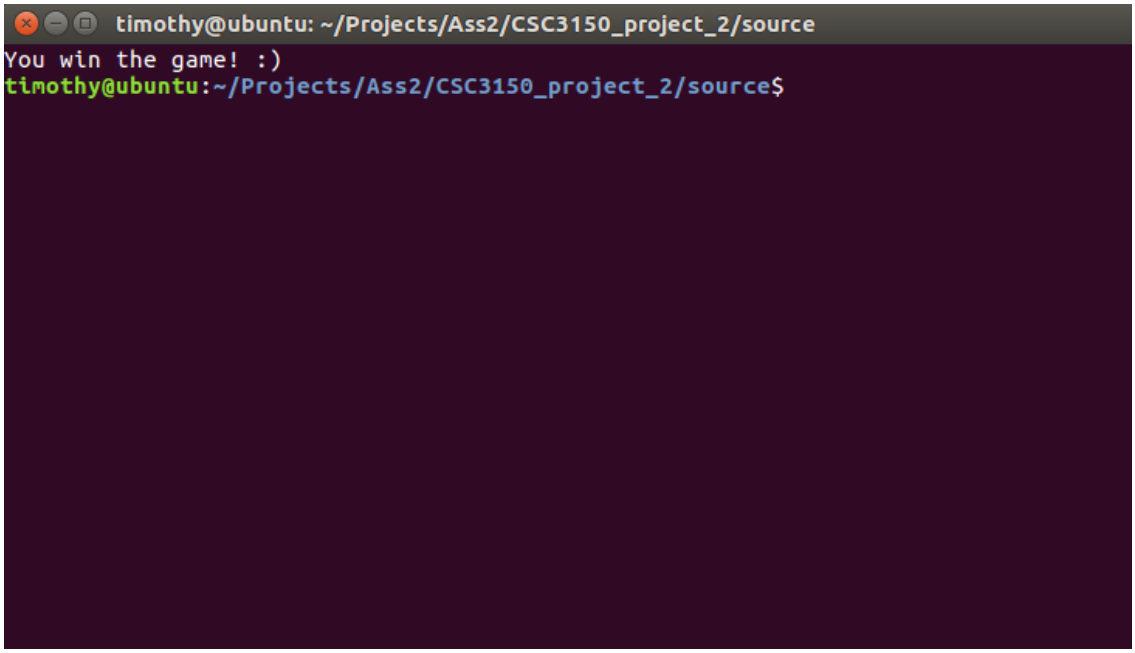
```
You win the game! :)
timothy@ubuntu:~/Projects/Ass2/CSC3150_project_2/source$
```

```
You lose the game! :(
timothy@ubuntu:~/Projects/Ass2/CSC3150_project_2/source$
```

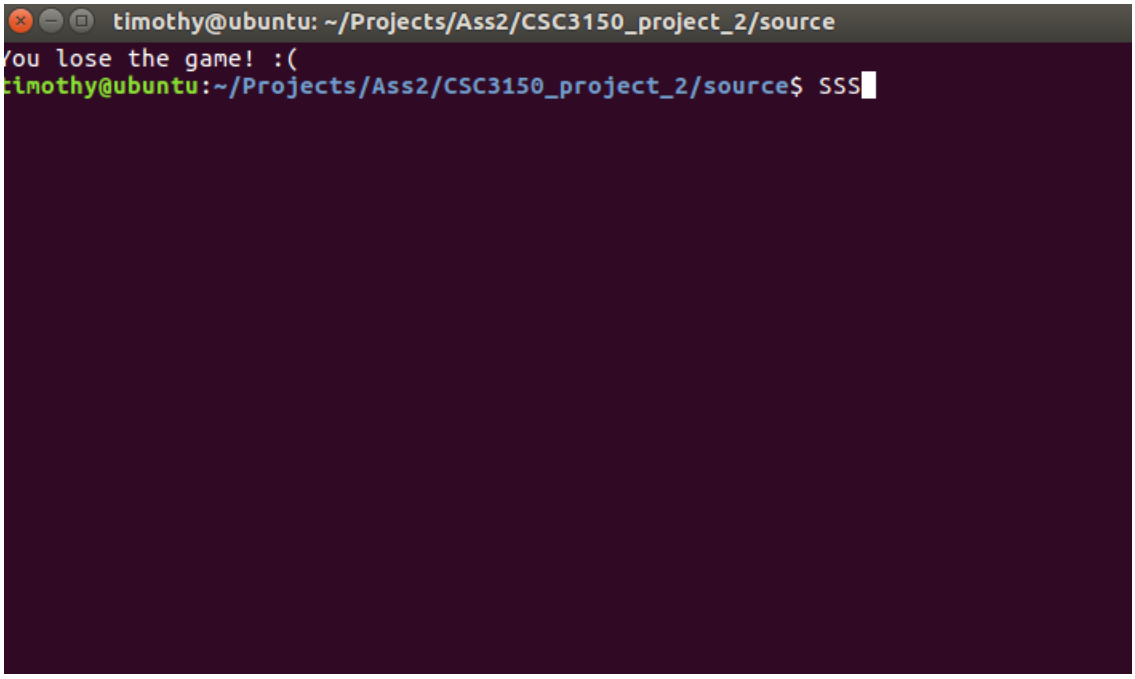
```
You quit the game!
timothy@ubuntu:~/Projects/Ass2/CSC3150_project_2/source$
```

- Screenshots of the Output

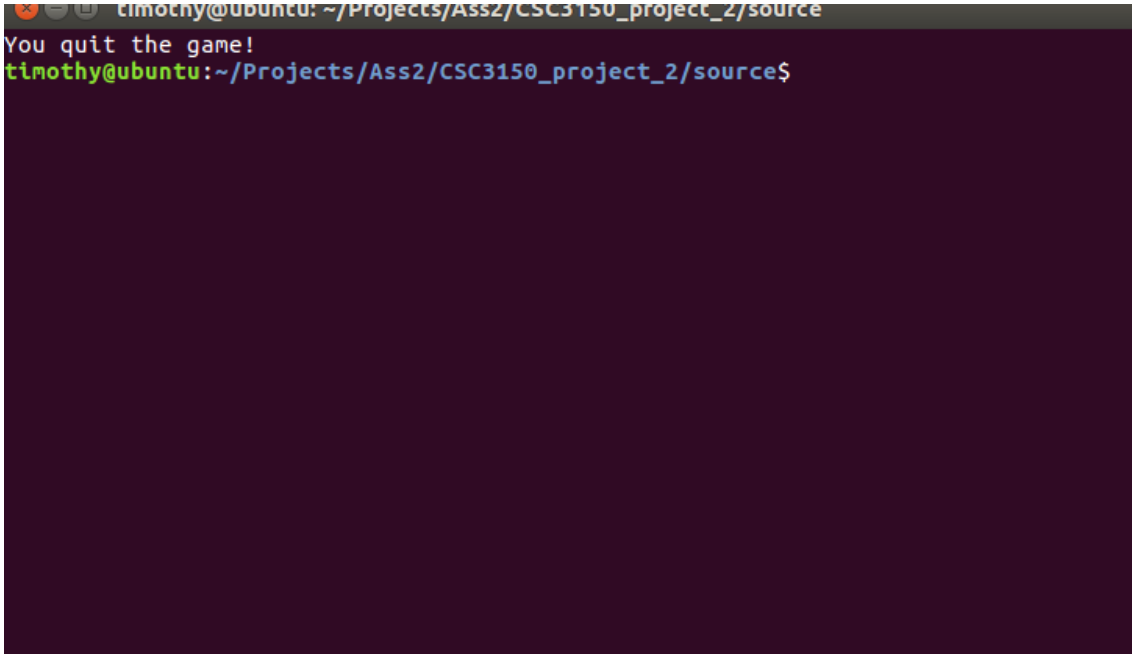
  Win


timothy@ubuntu: ~/Projects/Ass2/CSC3150_project_2/source
You win the game! :)
timothy@ubuntu:~/Projects/Ass2/CSC3150_project_2/source$

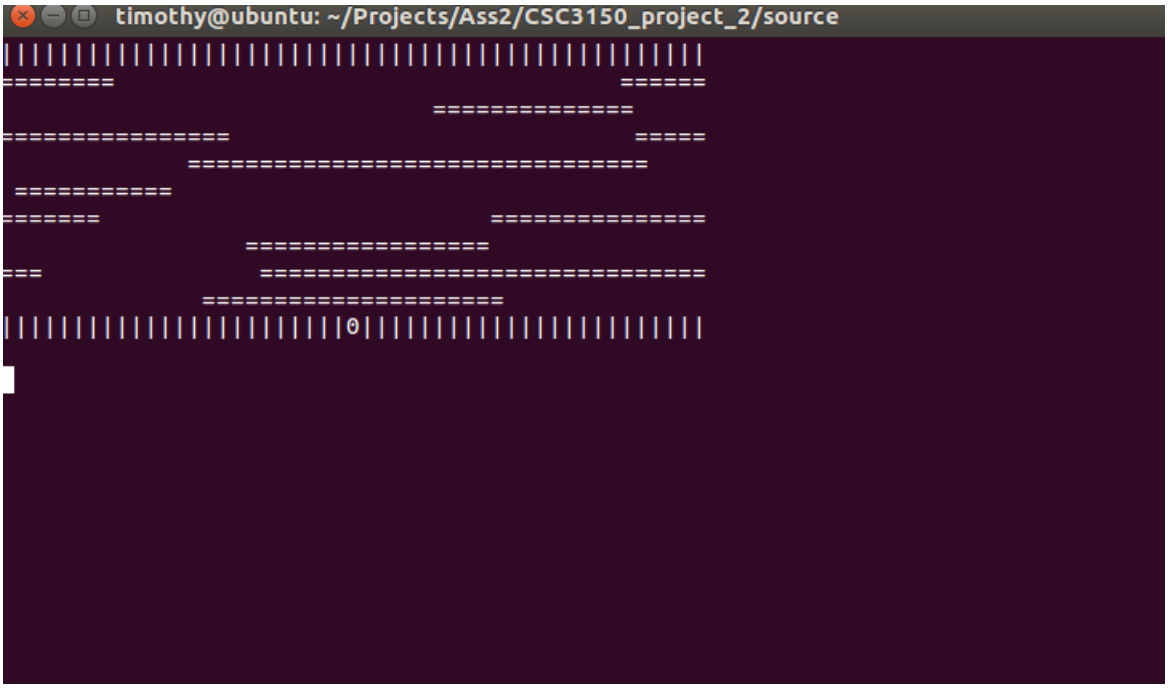
Lose


timothy@ubuntu: ~/Projects/Ass2/CSC3150_project_2/source
You lose the game! :(
timothy@ubuntu:~/Projects/Ass2/CSC3150_project_2/source$ SSS

Quit


You quit the game!
timothy@ubuntu:~/Projects/Ass2/CSC3150_project_2/source$

Play Mode



- Video of the game

  If you want to see the whole game process, you can refer to the demo videos which are included in the following link.

  https://share.weiyun.com/XJuPcLDP

Bonus

I only implement the second part of bonus task (the random log length). The log length is generated randomly by the rand() function.

In my code, I define several global variables to control the range of the length of the log, you may change them.

```
#define LOGLENGTH 10
// the base length of the log
#define LOGLENGTHVAR 25
// the variance of log length
// the actual log length = range(LOGLENGTH, LOGLENGTH+LOGLENGRHVAR)
```

Summary

The CSC3150 assignment 2 is about pthreads. Through implementing the frog game, I:

1. Use pthreads and have a better understanding of the threads and concurrency.
2. Read input key and display game in terminal, having more insights in controlling Linux terminal.
3. Implement a program with many functions, learning to organize the code.

4. Try to do the bonus task, learn about the graphics.h library (unfortunately I have not completed this part due to a strange bug)

The assignment helps me to have hand-on practice concerning the operation system.

Thanks for being patient to grade my code and read this boring report!