CSC3150 Project2 Report

118010045 Cui Yuncong

October 2020

Contents

1 Design

1.1 Program Flow

This program is written to implement the frog game with the multithread method — pthread. The program contains two threads which control the frog move and the log move separately.

At first, the program creates two pthreads. One is for calculating the frog move based on the keyboard input. The other is to print out logs and map. For the frog move part, use template and change the frog's position based on which key is pressed and save the new frog's position accordingly. For printing out logs, the another global array that stores starting cursor position of each line in the game map. After that, check the frog's position, and if the position is outside of boundary, ends the game and prints out the win or lose message. If the position of frog is in the river, which is blank space, then the game should end and print "You lose the game!!". On the other hand, if the position of the frog is equal to "=" which represents log, the game continues and frog and log shall move together to pre-defined direction.

1.2 Mutex Lock

The conflict can occur when the move of the frog intriguing by keyboard instructions and logs at the same time. Therefore, the mutex lock should be set in the both functions that processs keyboard instruction and log move.

To be more specific, the mutex locks is placed at the beginning of the while loop in the

frog move and the logs move functions. The mutex unlock command will be placed before the end of the while loop. This is because we want to send the signal in each iteration when the frog move function finished and let the logs move function to wait at the beginning until the frog move function terminates. This is because without the mutex lock the frog move and the logs move function will be executed simultaneously. They will change the map at the same time which may cause the unexpected error. With this mutex lock, the map will only be manipulated by one thread at a time. Thus, in order to prevent dead lock and guard the shared data, I have put the mutex lock where it needs to change the shared value, frog's position, which accessed and modified by both threads.

```
1  void* frog_move(void *idp){
2      int *my_id = (int*)idp;
3      char dir;
4      while (game_continue){
5          pthread_mutex_lock(&frog_mutex); // mutex lock
6          ......
7          pthread_cond_signal(&frog_threshold_cv);
8          pthread_mutex_unlock(&frog_mutex); // mutex unlock
9      }
10     pthread_cancel(threads[1]);
11     pthread_cond_signal(&frog_threshold_cv);
12     pthread_exit(NULL);
13 }
```

```
14 void* logs_move(void *idp){
15     int *my_id = (int*)idp;
16     int i, j;
17     while (game_continue){
18         pthread_mutex_lock(&frog_mutex); // mutex lock
19         pthread_cond_wait(&frog_threshold_cv, &frog_mutex);
20         ......
21         pthread_mutex_unlock(&frog_mutex); // mutex unlock
22     }
23     pthread_cancel(threads[0]);
24     pthread_exit(NULL);
25 }
```

2 Problem and Solution

First, I did not realize where I should put mutex lock. After figuring out the fact that mutex lock is for synchronizing between threads and for protecting the shared data. Then I realized that I need to use it for changing the frog's position.

Second, the time complexity of moving the logs each bit in each line is $O(n^2)$. However, I realized that the relative location of odd(even) lines stays the same. The move of logs in each line can be changed into moving the print start position. Thus, the time complexity can be reduced into $O(1)$.

3 Running Environment and Execution

Version of OS: Ubuntu 16.04.2

Kernel: Linux 4.10.14

```
[10/21/20]seed@VM:~$ sudo lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 16.04.2 LTS
Release:        16.04
Codename:       xenial
[10/21/20]seed@VM:~$ uname -srm
Linux 4.10.14 i686
```

Figure 1: Running Environment

Type “g++ hw2.cpp -lpthread” to compile the source code.

Type “./a.out” to run the game. Then the game is on. Use “wasd” to move frog and cross the river to win. If frog goes to outside of boundary, the game is over. If user types “q”, the program will exit. After the end, the output will tell the result.

```
[10/22/20]seed@VM:~/.../source$ g++ hw2.cpp -lpthread
[10/22/20]seed@VM:~/.../source$ ./a.out
```

Figure 2: Execution

4 Program Output

4.1 Beginning Stage

In the beginning, the frog steps on the river bank which is the bottom line in the map. “|” indicates bank. “=” indicates logs. “0” indicates frog. The empty space means the river.

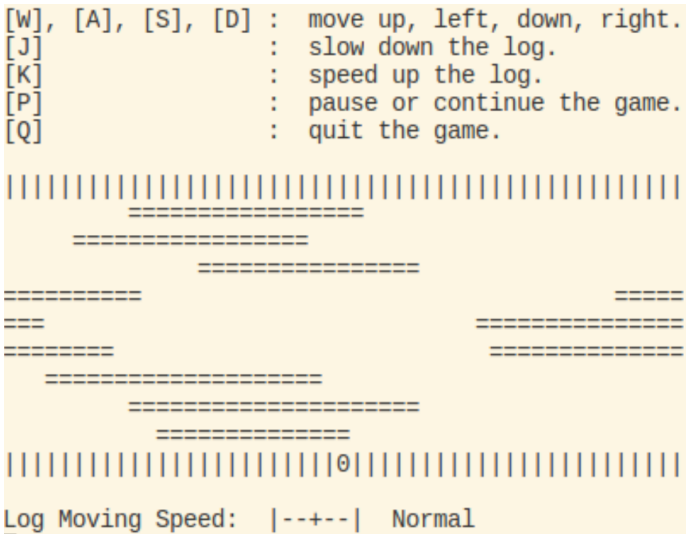

Figure 3: Beginning Stage

4.2 Running Stage

During the game, the frog steps on the logs which is the middle line in the map that consists of "=".

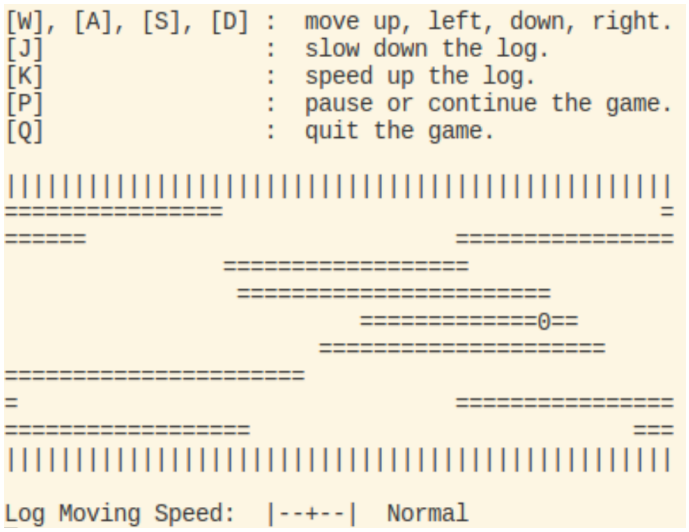

Figure 4: Running Stage

4.3 Result

The output "You win the game!!" which means the frog goes across the river.

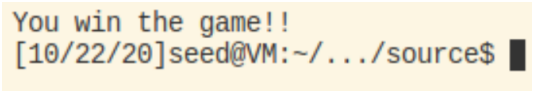

Figure 5: Winning Result

The output "You lose the game!!" which means the frog steps into the river or steps out of the boundary.

Figure 6: Lost Result

The output "You exit the game!!" which means the player enters Q/q key to quit the game.

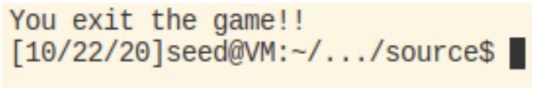

Figure 7: Exit Result

5 Bonus

5.1 Random Length of Logs

The length of each log is generated randomly and independently. However, the length of logs is longer than 1/4 of the length of the river and shorter than 1/2 of the length of the river to make sure the game is feasible to play.

```
1  void init(){
2      int i, j, s, l;
3      ......
4      srand((unsigned)time(NULL));
5      for(i = 1; i < ROW; i++){
6          s = rand() % (COLUMN - 1);
7          l = rand() % ((COLUMN - 1) / 4) + (COLUMN - 1) / 4;
8          for (j = s; j < s + l; j++)
9              map[i][j % (COLUMN - 1)] = '=';
10     }
11 }
```

5.2 Adjust the Speed of Logs

The speed of logs can be adjusted. The game has 5 gears, which are "Fastest", "Fast", "Normal", "Slow" and "Slowest". The user can use "j" to slow down logs or "k" to speed up logs.

```
Log Moving Speed:  |--+--|  Normal
```

Figure 8: "Normal" Model

```
Log Moving Speed:  |----+|  Fastest
```

Figure 9: "Fastest" Model

6 Learning Remarks

6.1 Process and Thread

A process, in the simplest terms, is an executing program. One or more threads run in the context of the process. A thread is the basic unit to which the operating system allocates processor time. A thread can execute any part of the process code, including parts currently being executed by another thread. There are differences between multi process and multi thread. Multi process is heavy and multi thread is light, moreover, one process can have multiple threads.

6.2 Create Multiple Threads

The pthread can be created by the pthread_create() function. If the thread is created successfully, the function will return 0. Otherwise the function will return error number.

6.3 The Thread Termination

The pthread can be terminated when the thread finishes its job or by the exit function. When the sub-thread terminates, it will not affect other sub-threads. However, without the exit function, if the main thread terminates, all the sub-thread will terminate automatically.

6.4 The Synchronization between Every Thread

We can use the pthread_join() function to realize the synchronization among all threads. The function will block all the threads until the threads terminate. The main function waits until other process is terminated using pthread_join(). Without pthread_join(), main

function will not wait for other pthreads and terminates. This will ensure that main thread will not execute the step next to the join function until the sub-threads exit.

6.5 Send and Receive Signals between Two Threads

We can use the pthread condition to send and receive the signal between two threads. Condition variable can be used in the function with mutex lock and will block the function with wait function until another thread send a signal. This enables the threads to be executed without conflict.

6.6 The Usage of Mutex Lock

Mutex lock is used to lock the code between mutex lock and mutex unlock. This method is used to only allow one thread to execute the code at a time. Mutex lock prevents shared data which is called critical section, and using mutex lock, we can also prevent dead lock situation.

6.7 Show the Dynamic Figure

The sleep() and usleep() function can control the refresh speed of the interface, with which we are able to do the dynamic figure. The input number must be positive integer and the unit for sleep() function is second, for usleep() function is micro second.