Assignment 2

CSC3150

Student Number: 117010032

Student Name: 陈雅茜

香 港 中 文 大 學（深 圳）
The Chinese University of Hong Kong, Shenzhen

Introduction: Environment and Steps to Compile

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. The program is implemented on the cpp language and is executable on the ubuntu (ubuntu version 16.04 and kernel version v4.8.0-36-generic) and mac system. User should compile the cpp file and execute the executable file.

I also provided the optimized version which includes ten threads and have nine threads to control the logs move function.

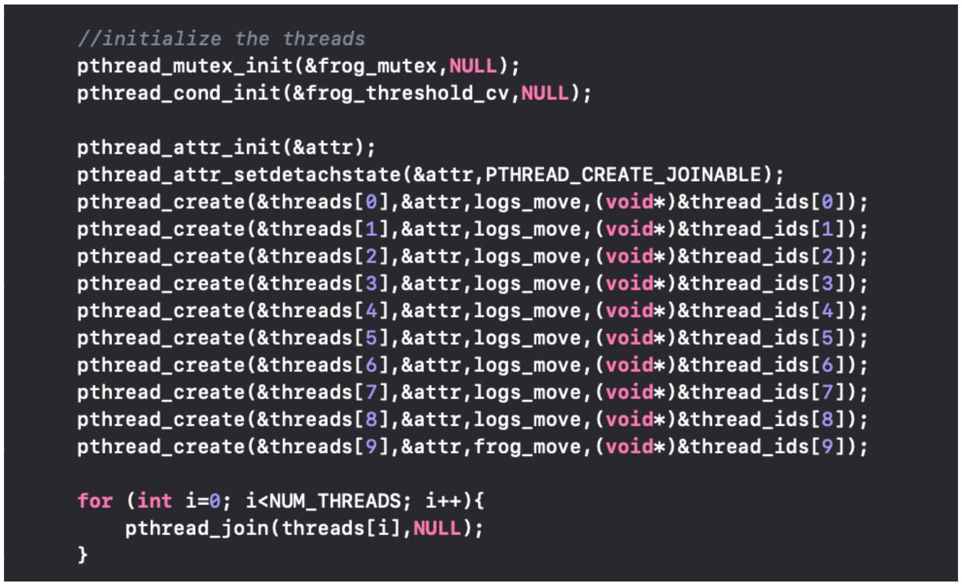
You should compile the file with command in your terminal under the source directory:

```
g++ hw2.cpp -lpthread
./a.out

g++ hw2_tenthreads.cpp -lpthread
./a.out
(for the version with ten threads)
```

The output will be you win if the frog successfully goes across the river.
The output will be you lose if the frog steps into the river or steps out of the boundary.
The output will be you quit if the player quit the game with the q key.

Flow: How Did I Design My Program

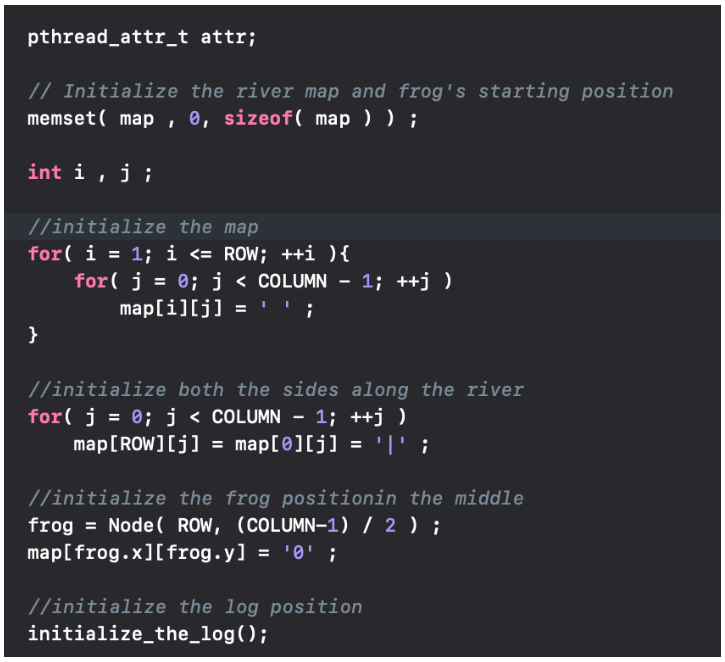
The flow for the two threads version and the ten threads version are very similar, except that the version with ten threads will execute the logs_move() function for nine times by nine different threads. The logs_move() function will send the signal to the frog_move() function after the 9th iteration to terminate the wait.



```
//initialize the threads
pthread_mutex_init(&frog_mutex,NULL);
pthread_cond_init(&frog_threshold_cv,NULL);

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);
pthread_create(&threads[0],&attr,logs_move,(void*)&thread_ids[0]);
pthread_create(&threads[1],&attr,logs_move,(void*)&thread_ids[1]);
pthread_create(&threads[2],&attr,logs_move,(void*)&thread_ids[2]);
pthread_create(&threads[3],&attr,logs_move,(void*)&thread_ids[3]);
pthread_create(&threads[4],&attr,logs_move,(void*)&thread_ids[4]);
pthread_create(&threads[5],&attr,logs_move,(void*)&thread_ids[5]);
pthread_create(&threads[6],&attr,logs_move,(void*)&thread_ids[6]);
pthread_create(&threads[7],&attr,logs_move,(void*)&thread_ids[7]);
pthread_create(&threads[8],&attr,logs_move,(void*)&thread_ids[8]);
pthread_create(&threads[9],&attr,frog_move,(void*)&thread_ids[9]);

for (int i=0; i<NUM_THREADS; i++){
    pthread_join(threads[i],NULL);
}
```

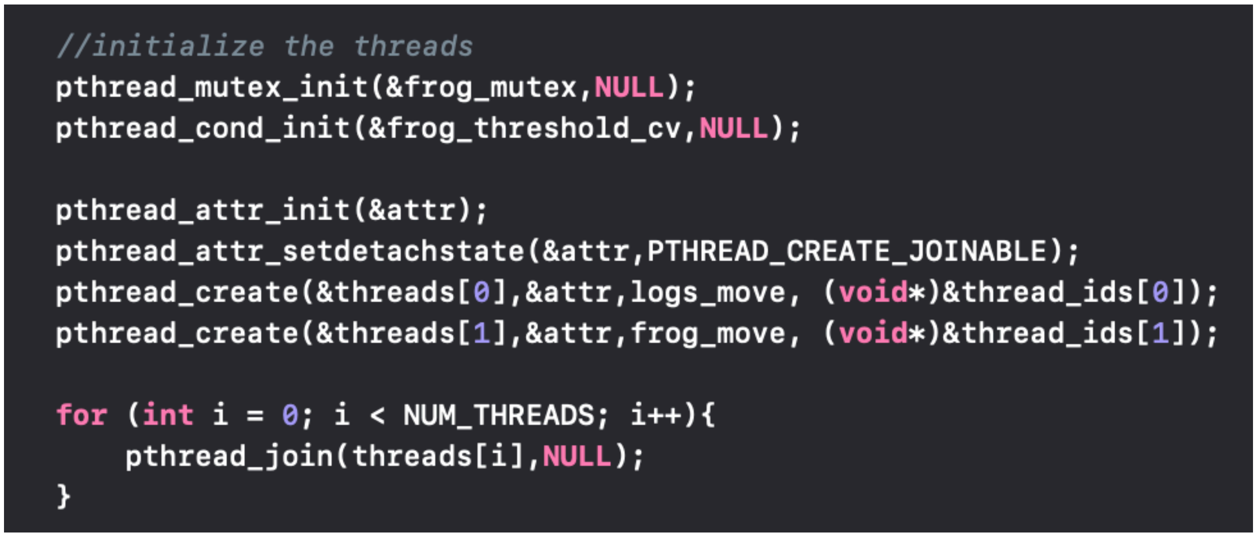
Below we show the flow for the two threads version.

(1) The program will first initialize the game and put the both sides along the river, all nine logs in the river and put the frog in the middle of the downward side.



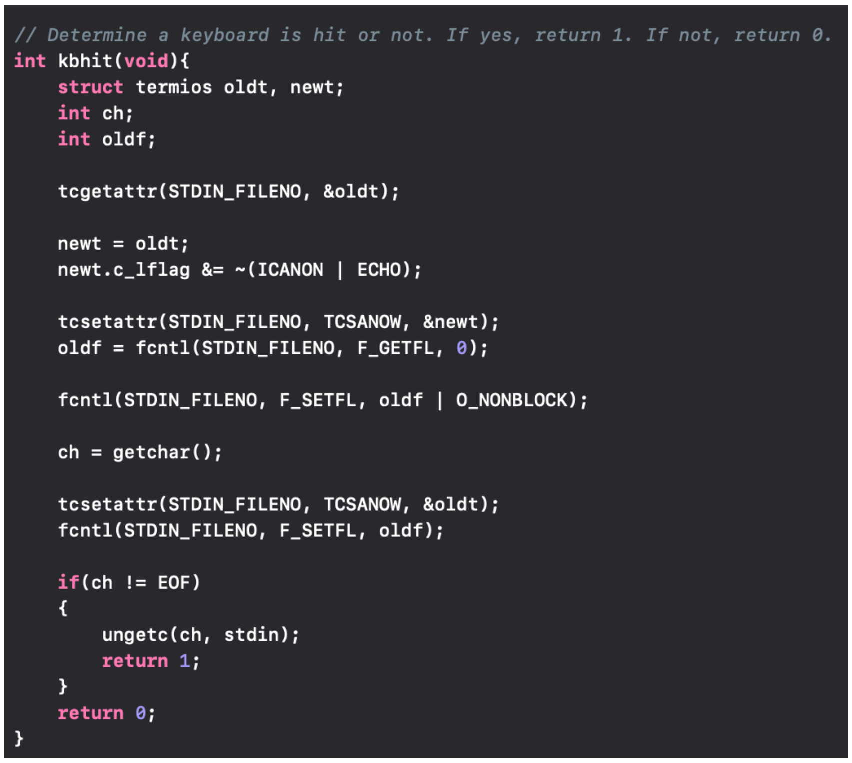
```
pthread_attr_t attr;

// Initialize the river map and frog's starting position
memset( map , 0, sizeof( map ) ) ;

int i , j ;

//initialize the map
for( i = 1; i <= ROW; ++i ){
    for( j = 0; j < COLUMN - 1; ++j )
        map[i][j] = ' ' ;
}

//initialize both the sides along the river
for( j = 0; j < COLUMN - 1; ++j )
    map[ROW][j] = map[0][j] = '|' ;

//initialize the frog positionin the middle
frog = Node( ROW, (COLUMN-1) / 2 ) ;
map[frog.x][frog.y] = '0' ;

//initialize the log position
initialize_the_log();
```

(2) Then the program will initialize two threads and mutex locks to control the frog move and the logs move separately.



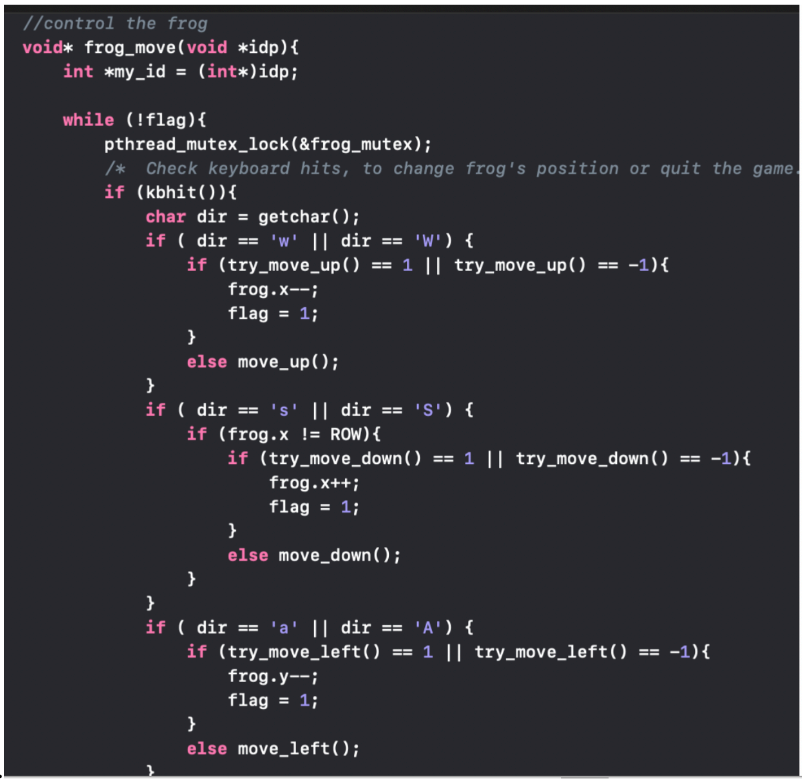
```
//initialize the threads
pthread_mutex_init(&frog_mutex,NULL);
pthread_cond_init(&frog_threshold_cv,NULL);

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);
pthread_create(&threads[0],&attr,logs_move, (void*)&thread_ids[0]);
pthread_create(&threads[1],&attr,frog_move, (void*)&thread_ids[1]);

for (int i = 0; i < NUM_THREADS; i++){
    pthread_join(threads[i],NULL);
}
```

(3) The thread which controls the frog move will execute the frog move function in each iteration. The thread will catch the keyboard hit by the kbhit function and move the frog correspondingly.



```c
// Determine a keyboard is hit or not. If yes, return 1. If not, return 0.
int kbhit(void){
    struct termios oldt, newt;
    int ch;
    int oldf;

    tcgetattr(STDIN_FILENO, &oldt);

    newt = oldt;
    newt.c_lflag &= ~(ICANON | ECHO);

    tcsetattr(STDIN_FILENO, TCSANOW, &newt);
    oldf = fcntl(STDIN_FILENO, F_GETFL, 0);

    fcntl(STDIN_FILENO, F_SETFL, oldf | O_NONBLOCK);

    ch = getchar();

    tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
    fcntl(STDIN_FILENO, F_SETFL, oldf);

    if(ch != EOF)
    {
        ungetc(ch, stdin);
        return 1;
    }
    return 0;
}
```

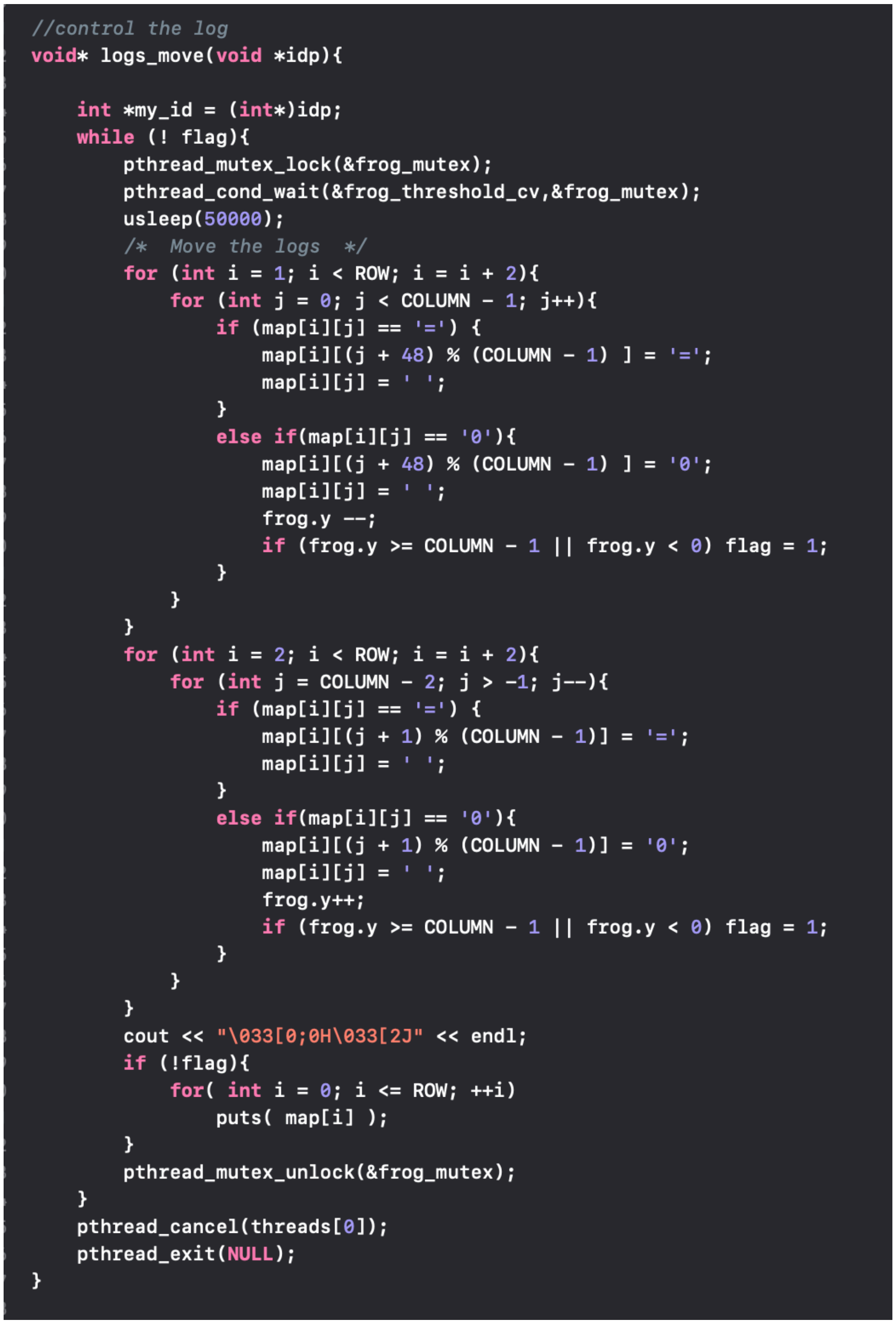
The thread will be unlocked after each iteration and send the signal to the log move function



```c
//control the frog
void* frog_move(void *idp){
    int *my_id = (int*)idp;

    while (!flag){
        pthread_mutex_lock(&frog_mutex);
        /*  Check keyboard hits, to change frog's position or quit the game.
        if (kbhit()){
            char dir = getchar();
            if ( dir == 'w' || dir == 'W') {
                if (try_move_up() == 1 || try_move_up() == -1){
                    frog.x--;
                    flag = 1;
                }
                else move_up();
            }
            if ( dir == 's' || dir == 'S') {
                if (frog.x != ROW){
                    if (try_move_down() == 1 || try_move_down() == -1){
                        frog.x++;
                        flag = 1;
                    }
                    else move_down();
                }
            }
            if ( dir == 'a' || dir == 'A') {
                if (try_move_left() == 1 || try_move_left() == -1){
                    frog.y--;
                    flag = 1;
                }
                else move_left();
            }
```

.

(4) Another thread will execute the log move function after receiving the signal which will control the log move. For the odd row it will move the log to left for one block. For the even row, it will move the log to the right for one block. Then it will put the map into the screen.



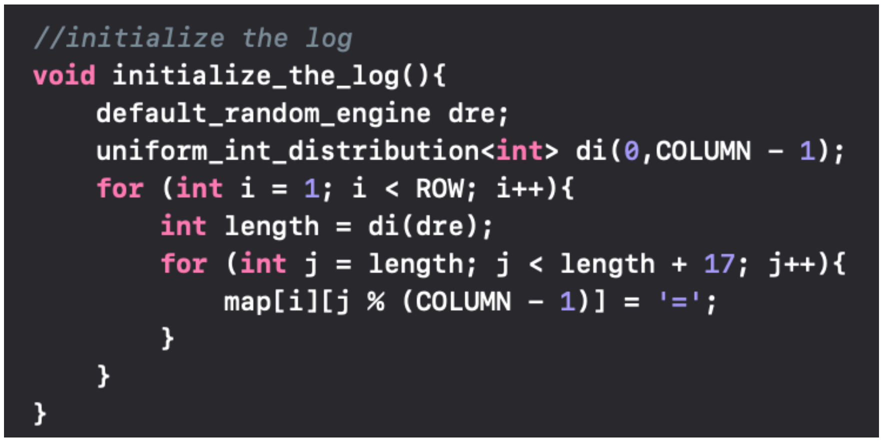
```
//control the log
void* logs_move(void *idp){

    int *my_id = (int*)idp;
    while (! flag){
        pthread_mutex_lock(&frog_mutex);
        pthread_cond_wait(&frog_threshold_cv,&frog_mutex);
        usleep(50000);
        /*  Move the logs  */
        for (int i = 1; i < ROW; i = i + 2){
            for (int j = 0; j < COLUMN - 1; j++){
                if (map[i][j] == '=') {
                    map[i][(j + 48) % (COLUMN - 1) ] = '=';
                    map[i][j] = ' ';
                }
                else if(map[i][j] == '0'){
                    map[i][(j + 48) % (COLUMN - 1) ] = '0';
                    map[i][j] = ' ';
                    frog.y --;
                    if (frog.y >= COLUMN - 1 || frog.y < 0) flag = 1;
                }
            }
        }
        for (int i = 2; i < ROW; i = i + 2){
            for (int j = COLUMN - 2; j > -1; j--){
                if (map[i][j] == '=') {
                    map[i][(j + 1) % (COLUMN - 1)] = '=';
                    map[i][j] = ' ';
                }
                else if(map[i][j] == '0'){
                    map[i][(j + 1) % (COLUMN - 1)] = '0';
                    map[i][j] = ' ';
                    frog.y++;
                    if (frog.y >= COLUMN - 1 || frog.y < 0) flag = 1;
                }
            }
        }
        cout << "\033[0;0H\033[2J" << endl;
        if (!flag){
            for( int i = 0; i <= ROW; ++i)
                puts( map[i] );
        }
        pthread_mutex_unlock(&frog_mutex);
    }
    pthread_cancel(threads[0]);
    pthread_exit(NULL);
}
```

(5) Repeat the steps from step 3 to step 4 until the player lose the game or win the game or quit the game.

Functions: How did I Design My Program & Where are My Autex Locks

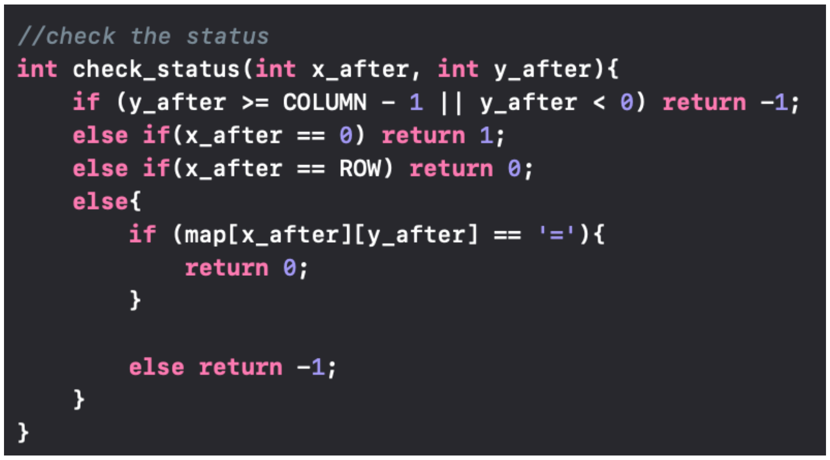
The functions in the ten-thread version and the two-thread version are exactly the same except that the logs_move() function is only for the log move in one row.

(1) initialize_the_log(): this function will initialize the log position in the map by random the first block position for each line. It will draw the 17-length log for each line.



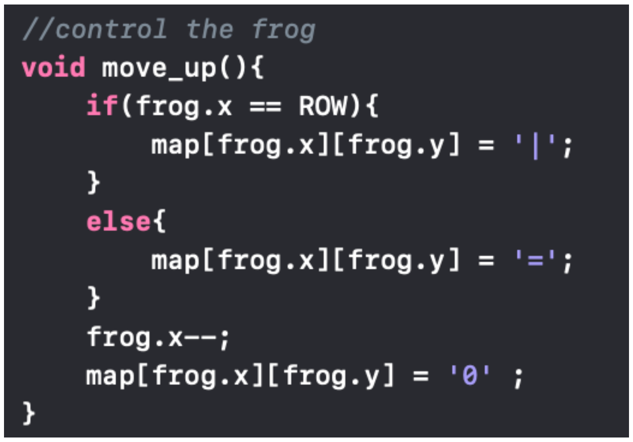
```
//initialize the log
void initialize_the_log(){
    default_random_engine dre;
    uniform_int_distribution<int> di(0,COLUMN - 1);
    for (int i = 1; i < ROW; i++){
        int length = di(dre);
        for (int j = length; j < length + 17; j++){
            map[i][j % (COLUMN - 1)] = '=';
        }
    }
}
```

(2) Check_status(int x_after, int y_after): this function will return the judgement that if the next movement of the frog is valid or not. If the frog will reach the top row (row 0), then the function will return 1. If the frog will drop into the river or go beyond the boundary, the function will return -1. Otherwise, the function will return 0.

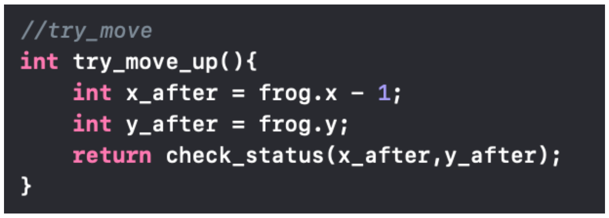


```
//check the status
int check_status(int x_after, int y_after){
    if (y_after >= COLUMN - 1 || y_after < 0) return -1;
    else if(x_after == 0) return 1;
    else if(x_after == ROW) return 0;
    else{
        if (map[x_after][y_after] == '='){
            return 0;
        }

        else return -1;
    }
}
```

(3) move_up() function will control the frog when it need to move up. It will change the position of frog and restore the position which was occupied by the frog. The same method with the functions move_down(), move_left() and move_right().

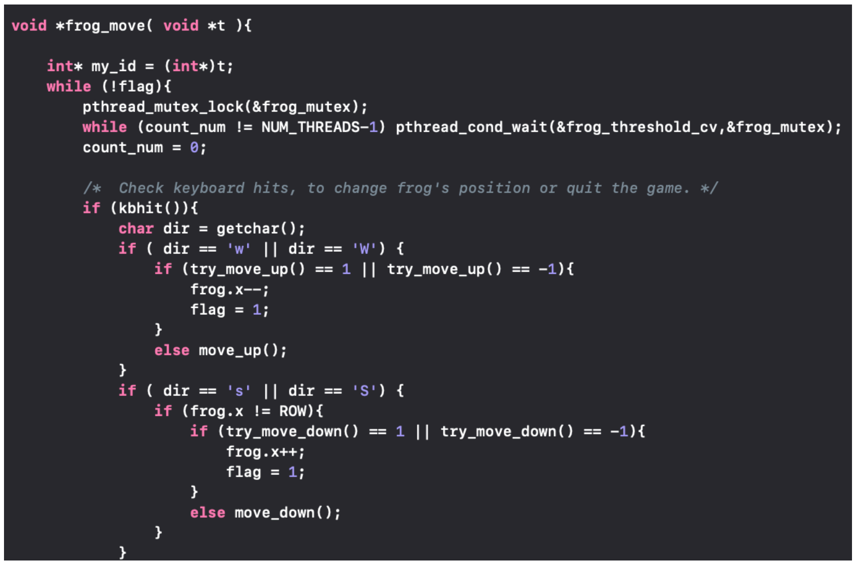

```c
//control the frog
void move_up(){
    if(frog.x == ROW){
        map[frog.x][frog.y] = '|';
    }
    else{
        map[frog.x][frog.y] = '=';
    }
    frog.x--;
    map[frog.x][frog.y] = '0' ;
}
```

(4) Try_move_up() function will check if the next step is value or not by sending the next position of the frog to the check_status() function. The same method with try_move_down(), try_move_left() and try_move_right() functions.

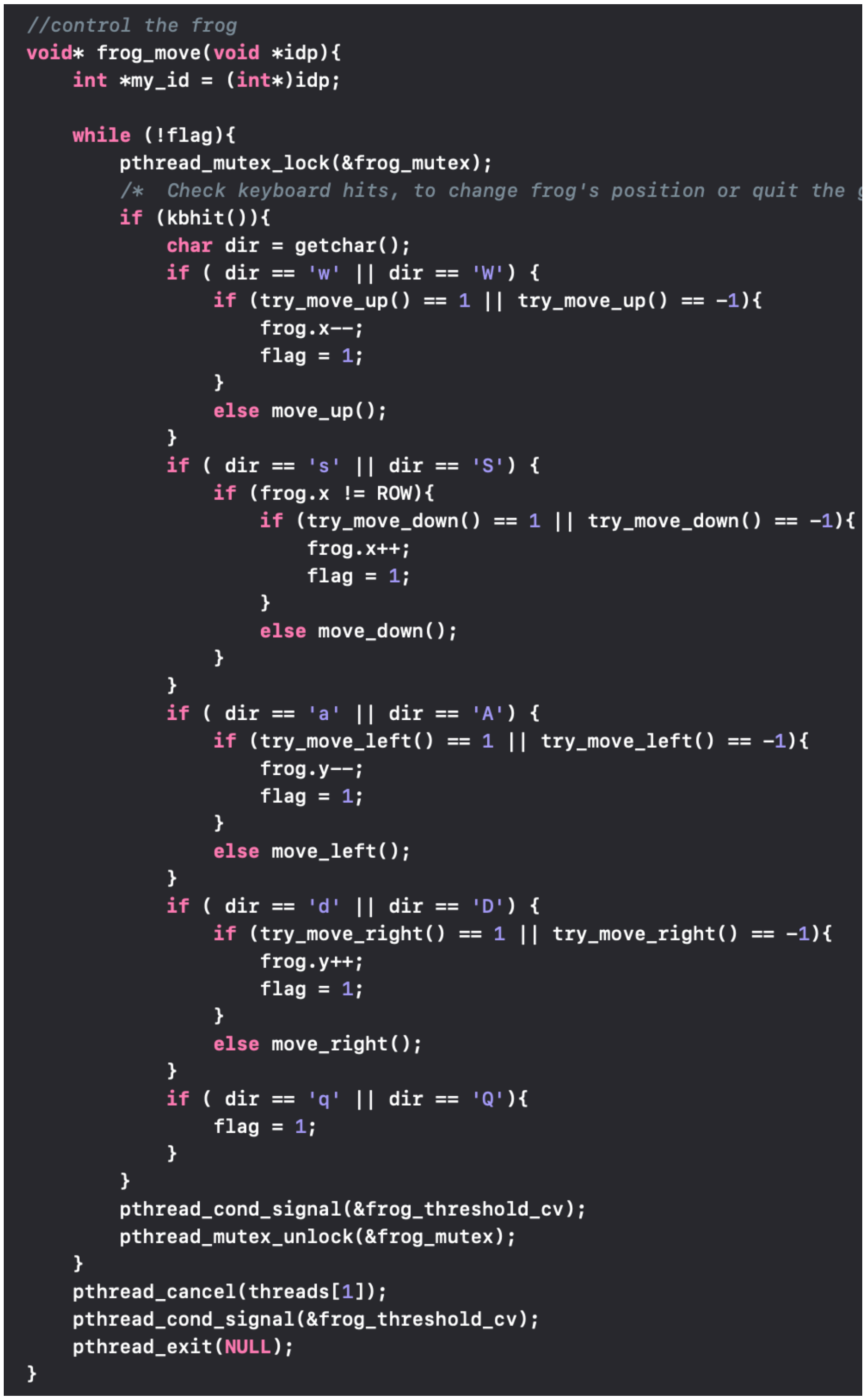

```c
//try_move
int try_move_up(){
    int x_after = frog.x - 1;
    int y_after = frog.y;
    return check_status(x_after,y_after);
}
```

(5) frog_move_function() is used to control the frog move when the program catch the keyboard hit signal. The whole function is executed in a sub-thread. The function will also send the signal to the logs_move_function() to terminate the function waiting. The mutex lock will be placed here since we need this function to send the signal to the logs_move function.

Function for the ten-thread version


```c
void *frog_move( void *t ){

    int* my_id = (int*)t;
    while (!flag){
        pthread_mutex_lock(&frog_mutex);
        while (count_num != NUM_THREADS-1) pthread_cond_wait(&frog_threshold_cv,&frog_mutex);
        count_num = 0;

        /*  Check keyboard hits, to change frog's position or quit the game. */
        if (kbhit()){
            char dir = getchar();
            if ( dir == 'w' || dir == 'W') {
                if (try_move_up() == 1 || try_move_up() == -1){
                    frog.x--;
                    flag = 1;
                }
                else move_up();
            }
            if ( dir == 's' || dir == 'S') {
                if (frog.x != ROW){
                    if (try_move_down() == 1 || try_move_down() == -1){
                        frog.x++;
                        flag = 1;
                    }
                    else move_down();
                }
            }
```

Function for the two-thread version

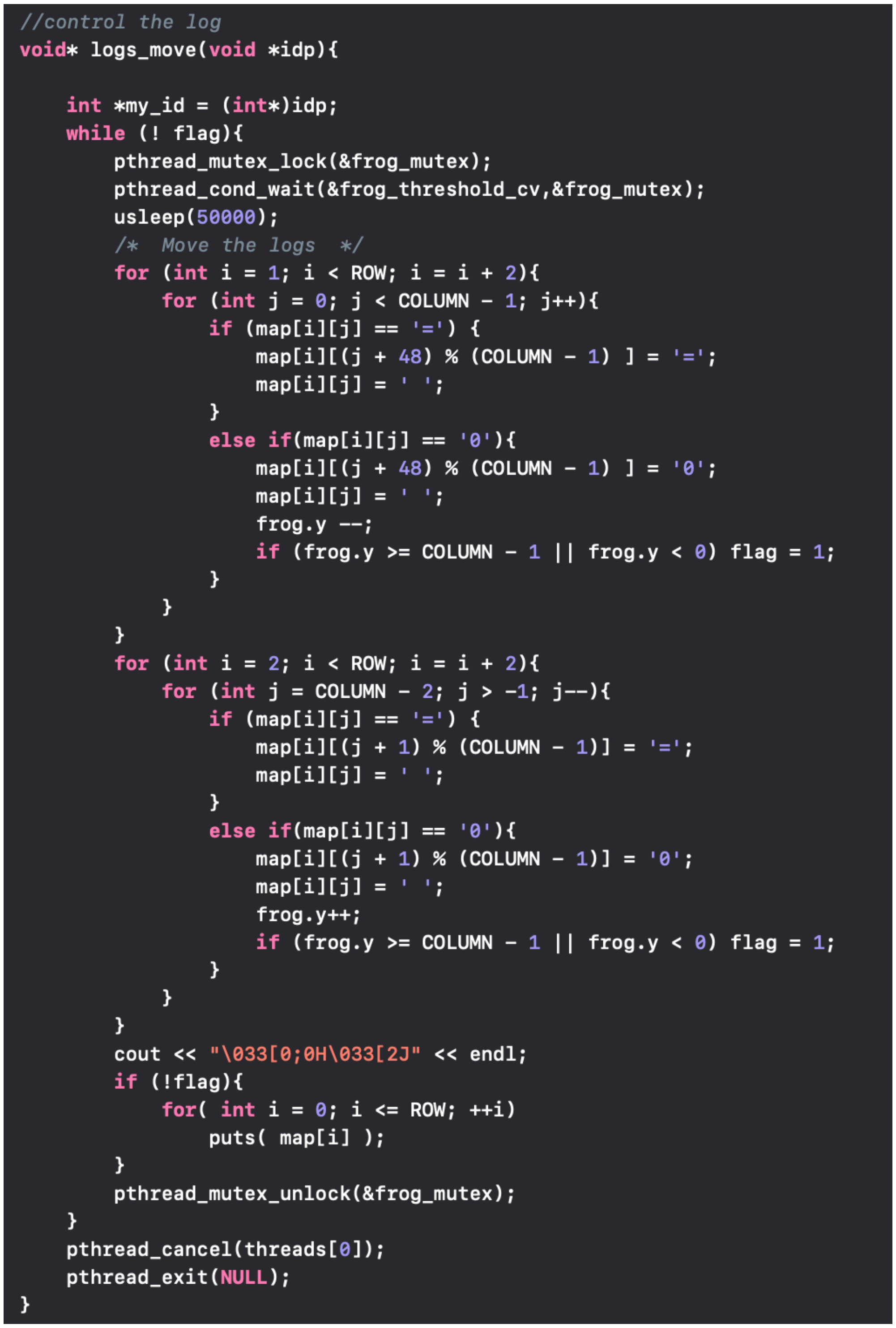


```c
//control the frog
void* frog_move(void *idp){
    int *my_id = (int*)idp;

    while (!flag){
        pthread_mutex_lock(&frog_mutex);
        /*  Check keyboard hits, to change frog's position or quit the
        if (kbhit()){
            char dir = getchar();
            if ( dir == 'w' || dir == 'W') {
                if (try_move_up() == 1 || try_move_up() == -1){
                    frog.x--;
                    flag = 1;
                }
                else move_up();
            }
            if ( dir == 's' || dir == 'S') {
                if (frog.x != ROW){
                    if (try_move_down() == 1 || try_move_down() == -1){
                        frog.x++;
                        flag = 1;
                    }
                    else move_down();
                }
            }
            if ( dir == 'a' || dir == 'A') {
                if (try_move_left() == 1 || try_move_left() == -1){
                    frog.y--;
                    flag = 1;
                }
                else move_left();
            }
            if ( dir == 'd' || dir == 'D') {
                if (try_move_right() == 1 || try_move_right() == -1){
                    frog.y++;
                    flag = 1;
                }
                else move_right();
            }
            if ( dir == 'q' || dir == 'Q'){
                flag = 1;
            }
        }
        pthread_cond_signal(&frog_threshold_cv);
        pthread_mutex_unlock(&frog_mutex);
    }
    pthread_cancel(threads[1]);
    pthread_cond_signal(&frog_threshold_cv);
    pthread_exit(NULL);
}
```

(6) logs_move() function is used to control the logs move. In each iteration the log including the frog on the block will move for one block. This function will check if the frog is out of the boundary or not. The function will also wait for the signal in the beginning of the

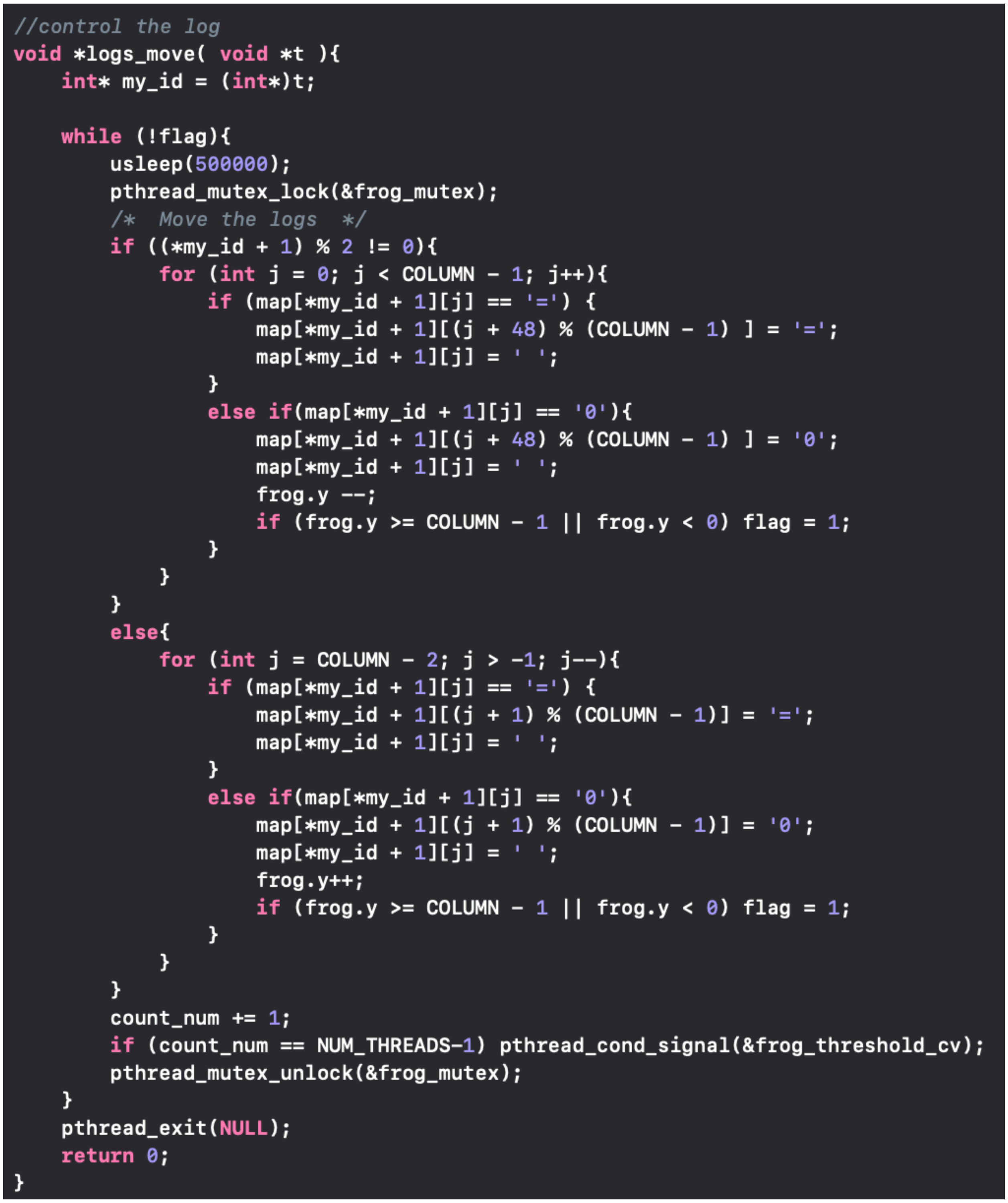
while loop. The mutex lock will be placed here since we need this function to wait for the signal from the frog_move() function.

Function for the two-thread version



```
//control the log
void* logs_move(void *idp){

    int *my_id = (int*)idp;
    while (! flag){
        pthread_mutex_lock(&frog_mutex);
        pthread_cond_wait(&frog_threshold_cv,&frog_mutex);
        usleep(50000);
        /*  Move the logs  */
        for (int i = 1; i < ROW; i = i + 2){
            for (int j = 0; j < COLUMN - 1; j++){
                if (map[i][j] == '=') {
                    map[i][(j + 48) % (COLUMN - 1) ] = '=';
                    map[i][j] = ' ';
                }
                else if(map[i][j] == '0'){
                    map[i][(j + 48) % (COLUMN - 1) ] = '0';
                    map[i][j] = ' ';
                    frog.y --;
                    if (frog.y >= COLUMN - 1 || frog.y < 0) flag = 1;
                }
            }
        }
        for (int i = 2; i < ROW; i = i + 2){
            for (int j = COLUMN - 2; j > -1; j--){
                if (map[i][j] == '=') {
                    map[i][(j + 1) % (COLUMN - 1)] = '=';
                    map[i][j] = ' ';
                }
                else if(map[i][j] == '0'){
                    map[i][(j + 1) % (COLUMN - 1)] = '0';
                    map[i][j] = ' ';
                    frog.y++;
                    if (frog.y >= COLUMN - 1 || frog.y < 0) flag = 1;
                }
            }
        }
        cout << "\033[0;0H\033[2J" << endl;
        if (!flag){
            for( int i = 0; i <= ROW; ++i)
                puts( map[i] );
        }
        pthread_mutex_unlock(&frog_mutex);
    }
    pthread_cancel(threads[0]);
    pthread_exit(NULL);
}
```

Function for the ten-thread version



```c
//control the log
void *logs_move( void *t ){
    int* my_id = (int*)t;

    while (!flag){
        usleep(500000);
        pthread_mutex_lock(&frog_mutex);
        /*  Move the logs  */
        if ((*my_id + 1) % 2 != 0){
            for (int j = 0; j < COLUMN - 1; j++){
                if (map[*my_id + 1][j] == '=') {
                    map[*my_id + 1][(j + 48) % (COLUMN - 1) ] = '=';
                    map[*my_id + 1][j] = ' ';
                }
                else if(map[*my_id + 1][j] == '0'){
                    map[*my_id + 1][(j + 48) % (COLUMN - 1) ] = '0';
                    map[*my_id + 1][j] = ' ';
                    frog.y --;
                    if (frog.y >= COLUMN - 1 || frog.y < 0) flag = 1;
                }
            }
        }
        else{
            for (int j = COLUMN - 2; j > -1; j--){
                if (map[*my_id + 1][j] == '=') {
                    map[*my_id + 1][(j + 1) % (COLUMN - 1)] = '=';
                    map[*my_id + 1][j] = ' ';
                }
                else if(map[*my_id + 1][j] == '0'){
                    map[*my_id + 1][(j + 1) % (COLUMN - 1)] = '0';
                    map[*my_id + 1][j] = ' ';
                    frog.y++;
                    if (frog.y >= COLUMN - 1 || frog.y < 0) flag = 1;
                }
            }
        }
        count_num += 1;
        if (count_num == NUM_THREADS-1) pthread_cond_signal(&frog_threshold_cv);
        pthread_mutex_unlock(&frog_mutex);
    }
    pthread_exit(NULL);
    return 0;
}
```

Where are My Autex Locks

For the two-thread version

As mentioned the previous part, the mutex locks will be 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.

For the ten-thread version

The position to place the mutex lock is exactly the same as the two-thread version. However, for ten-thread version, the function to send the signal is placed in the logs_move() function and the wait function is placed at the beginning of the frog_move() function. There is also another reason for the ten-thread version to have mutex lock in the logs_move() function. Since there are multiple threads which will execute the function, we need to ensure that there is only one thread executing the function at the time.

The Problems I Met in the Assignment and Solutions

(1) If the execution is only within the while loop, the program will iterate too fast which is not only bad for the player but also will cause screen flickering. This solution for this problem is to add a usleep() function in the logs_move() function. It will extend the time for one iteration and make the interface relatively stable.

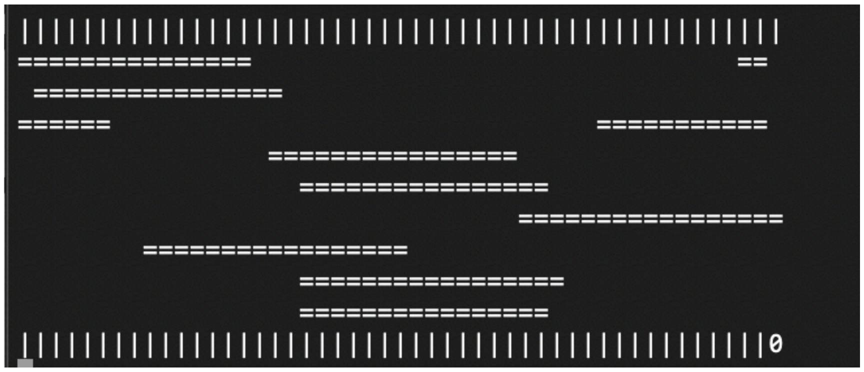
(2) When the frog_move() function detected the game end and break the while loop. The other thread for logs_move() function will not terminate automatically. It will wait for the signal from the frog_move() forever or cause the dead loop if add the send signal function before the exit in the frog_move() function. The same problem will occur if the logs_move() function terminates before. I also tried to use the thread cancel function in order to terminate the other thread before one thread ends. However, this solution does not work. The final solution is to add a condition check in the while loop and declare a global variable flag to check if the game ended or not. If the game ends, the flag will be 1 and the while loop will terminate automatically.

(3) I initially put the wait signal function in the middle of the move log loop, only before puts function. The game will occasionally exit unexpectedly since the map is manipulated by the two threads simultaneously. The solution is put the wait signal function at the beginning of the move log function.

Results: Screen Shot of My Output Result

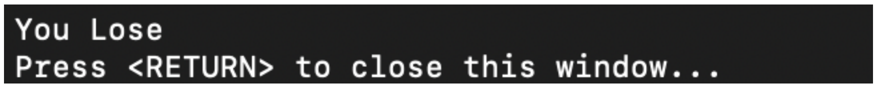
The result output are exactly the same with two versions.

(1) Check the right boundary when frog is at the down side of the river



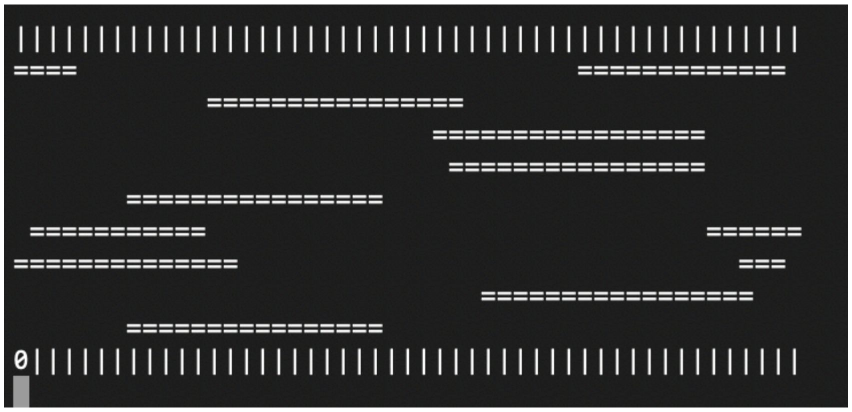
```
|||||||||||||||||||||||||||||||||||||||||||||||||
===============                               ==
 ================
======                               ===========
                ================
                  ================
                                =================
        =================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

The result for this condition



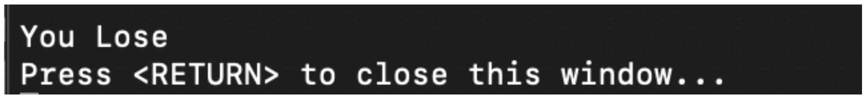
```
You Lose
Press <RETURN> to close this window...
```

(2) Check the left boundary when frog is at the down side of the river



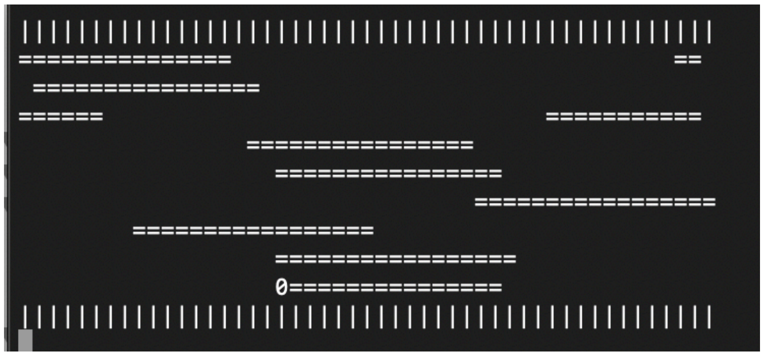
```
|||||||||||||||||||||||||||||||||||||||||||||||||
====                               =============
            ================
                          =================
                           ================
       ================
 ===========                               ======
==============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

The result for this condition



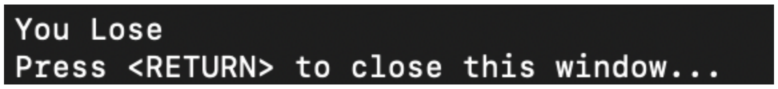
```
You Lose
Press <RETURN> to close this window...
```

(3) Check the condition when frog jump off the log from the left side in the odd number row



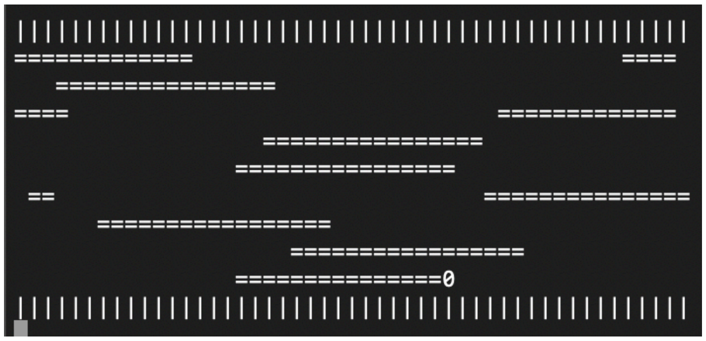
```
|||||||||||||||||||||||||||||||||||||||||||||||||
===============                               ==
 ================
======                               ===========
                ================
                  ================
                                =================
        =================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

The result for this condition



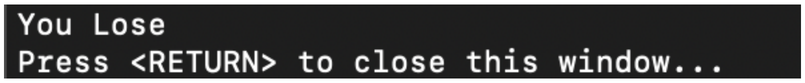
```
You Lose
Press <RETURN> to close this window...
```

(4) Check the condition when the frog jump off the log from the right side in the odd number row



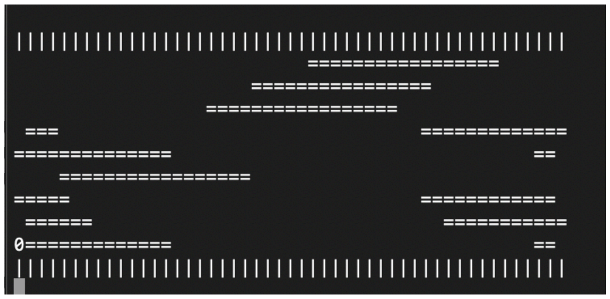
```
|||||||||||||||||||||||||||||||||||||||||||||||||
=============                               ====
   ================
====                               =============
                  ================
                ================
 ==                               ===============
      =================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

The result for this condition



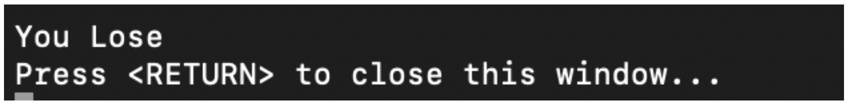
```
You Lose
Press <RETURN> to close this window...
```

(5) Check the condition when the frog go beyond the boundary with the log on the left side



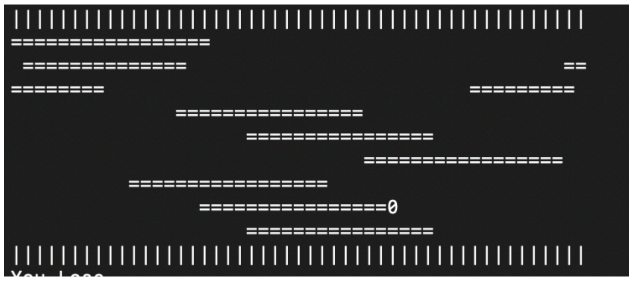
```
|||||||||||||||||||||||||||||||||||||||||||||||||
                          =================
                     ================
                 =================
 ===                                =============
==============                                ==
    =================
=====   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

The result for this condition is



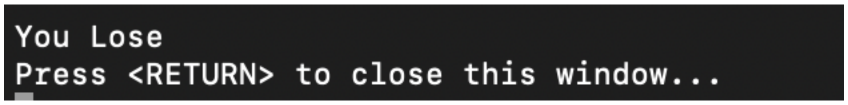
```
You Lose
Press <RETURN> to close this window...
```

(6) Check the condition when frog jump off the log from the right side in the even number row



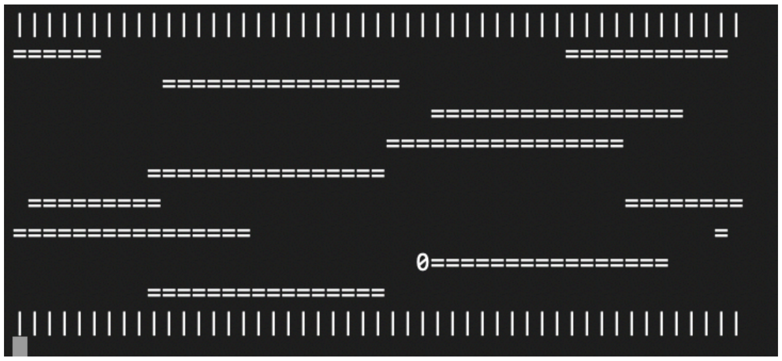
```
|||||||||||||||||||||||||||||||||||||||||||||||||
=================
 ==============                                ==
========                               =========
              ================
                    ================
                              =================
          =================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

The result for this condition



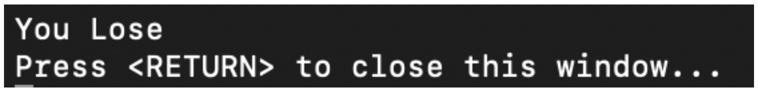
```
You Lose
Press <RETURN> to close this window...
```

(7) Check the condition when frog jump off the log from the left side in the even number row



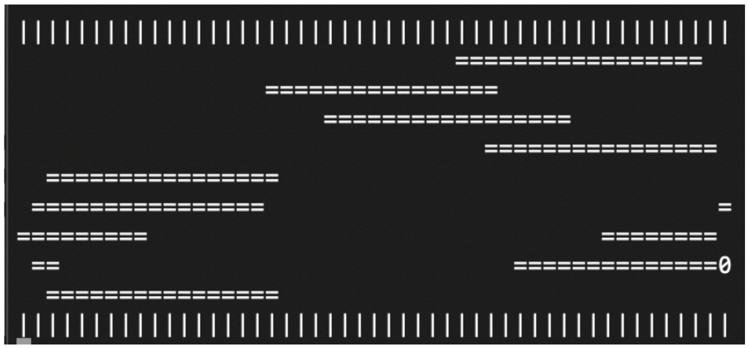
```
|||||||||||||||||||||||||||||||||||||||||||||||||
======                               ===========
          ================
                            =================
                         ================
         ================
 =========                               ========
================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

The result for this condition



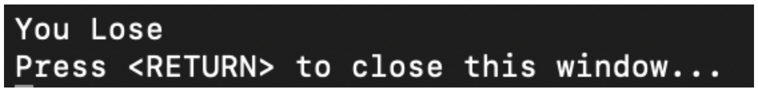
```
You Lose
Press <RETURN> to close this window...
```

(8) Check the condition when the frog go beyond the boundary with the log on the right side



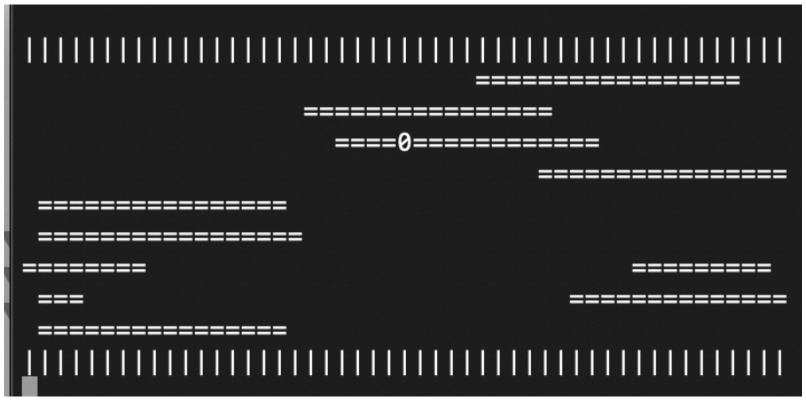
```
|||||||||||||||||||||||||||||||||||||||||||||||||
                              =================
                 ================
                     =================
                                ================
  ================
 ================                               =
=========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

The result for this condition is



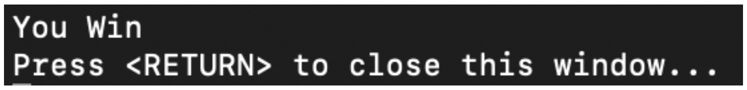
```
You Lose
Press <RETURN> to close this window...
```

(9) Check the condition when win



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

The result for this condition is



```
You Win
Press <RETURN> to close this window...
```

Conclusion: What I have learnt from this program

In this program assignment I learnt 1) how to create multiple threads 2) how do the thread terminate 3) how to realize the synchronization between every thread 4) how to send signals and receive the signals between two threads 5) how to use mutex lock to lock the function 6) how to use the sleep function to show the dynamic figure

(1) The p-thread 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.

(2) The p-thread 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.

(3) We can use the p-thread join function to realize the synchronization among all threads. The function will block all the threads until the threads terminate. This will ensure that main thread will not execute the step next to the join function until the sub-threads exit.

(4) We can use the p-thread 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.

(5) 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.

(6) We can use the sleep() and usleep() function to 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 method is second, for usleep method is micro second.