# CSC3150 Assignment1 Report

**Bonus and main task are both implemented. You can refer to them in the directory.**

# Part 0: Student Information

Student ID: 120090575
Student Name: 杜五洲DuWuzhou

# Part 1: Assignment Overview

This assignment includes two tasks: a frog game and bonus.

## 1.1 Frog Game Overview

The first task is the main task, which is a **multithreaded game**. This game is about a frog trying to cross the river. When the game begins, the frog will stand in the middle at bottom bank of river.

### 1.1.2 Keymap

1. The player can use **key "w"** to move up, **key "s"** to move own, **key "a"** to move left and **key "d"** to move right. Every key pressed down will make frog move **only one step away**.
2. The player can use **key "q"** to quit the game at any time.

### 1.1.3 How to Win or Lose

1. Only if the frog make it to the other bank of river the player wins.
   Between the two banks, there are many moving "woods", where the frog can stand on. Frog can utilize the woods to cross the river and finally make it to the other bank.
2. Player will lose if the frog falls to the river, i.e. the frog doesn't stand on the wood, or the frog hits the border of the screen.

### 1.1.4 Implementation

1. In the main thread, **initialize the map** representing all the contents in the screen. It also creates all the **child threads** in charge of the **woods display**. Then main thread detects whether the game can exit according to the **game status**. If not exiting, detect the **keyboard input** and continue to detect game status. Otherwise, all other child threads will be **joined** to the main thread and print the corresponding result of this game.
2. In the child thread, i.e. the `logs_move` function, every thread will generate the initial woods in random place, but in a fixed-length and continuous manner (no separation of different woods and only one wood per line) print only one line in the map. What's more, each thread will be responsible for **monitoring** whether the frog hits the border or falls to the river in its line, so as to timely **updating the game status**. In addition, if the game continues, the child thread will **update the map** every 0.5s to simulate the woods' and frog's moving.

## 1.2 Bonus

The second task is to implement a **thread pool** with fixed number of threads and with a handler function receiving an `int` parameter.

### 1.2.2 Idea and Design

1. The function `async_init` is to initialize the thread pool. The thread pool should include two main attributes **a thread array (or just a number of threads) to handle multiple tasks and a task queue to be handled**. In order to arrange the tasks in the task queue to different threads, a **mutex lock** is needed to prevent duplicate arrangement and some synchronization problems. The child thread should wait until the queue has tasks, so a **thread condition** is needed to wake up the child thread.
2. The function `async_run` implements the **task addition of task queue** in the thread pool.
3. The function `routine_fun` is the child thread function. It shows how the child thread **firstly sleeps** until a task enters and **wakes the thread to handle the task**.

# Part 2: Implementation Details

## 2.1 Main Task

### 2.1.1 Global Variables and Macros

```c
#define ROW 10
#define COLUMN 50
#define LOGLEN 15


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


char map[ROW+10][COLUMN] ;

// pthread_mutex_t status_lock;
pthread_mutex_t frog_lock;
pthread_mutex_t map_lock;
int status = 0;
```

The frog struct is just to represent which row and column the frog is in. There are actually 11 rows in the frog game, representing 2 banks and 9 lines of woods. There are two **mutex locks** to handle the protection for `frog` and `map` access and modification. No lock for status since **as long as changing the status all threads will just quit and join the main thread**, according to the game regulations. So, there is **no synchronization problem** for status. Initially, the status is 0, representing the **game continuing status** and no need to exit.

## 2.1.2 Whole Trace

```c
int main( int argc, char *argv[] ){

    // Initialize the river map and frog's starting position
    printf("\033[2J\033[H\033[?25l");
    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] = '|' ;

    for( j = 0; j < COLUMN - 1; ++j )
        map[0][j] = map[0][j] = '|' ;

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

    //Print the map into screen
    for( i = 0; i <= ROW; ++i)
        puts( map[i] );

    if (pthread_mutex_init(&frog_lock, NULL)){
        printf("frog_lock init fails\n");
        exit(1);
    }

    if (pthread_mutex_init(&map_lock, NULL)){
        printf("map_lock init fails\n");
        exit(1);
    }

    pthread_t threads[ROW+1];
    int thread_create_status;
    for ( i = 0; i < ROW+1; ++i){
        thread_create_status = pthread_create(&threads[i], NULL, logs_move, (void*) (intptr_t)i);
        if(thread_create_status){
            printf("ERROR: return status from pthread_create: %d", thread_create_status);
            exit(1);
        }
    }
```

Let's trace the `main` function.

Firstly, map should be initialized according to the game starting state, frog in the middle of bottome bank and no woods exist. Then initialize all the **mutex locks** and create all the 11 child threads.

```
while(!status){
    if (kbhit()){
        char input = getchar();
        if (input == 'q' || input == 'Q'){
            // pthread_mutex_lock(&status_lock);
            status = 3;
            // pthread_mutex_unlock(&status_lock);
        }
        else if (input == 'w' || input == 'W'){
            pthread_mutex_lock(&frog_lock);
            --frog.x;
            pthread_mutex_unlock(&frog_lock);

            pthread_mutex_lock(&map_lock);
            map[frog.x][frog.y] = '0';
            map[frog.x+1][frog.y] = (frog.x+1 == 10) ? '|' : '=';
            pthread_mutex_unlock(&map_lock);

            // printf("after type in w, frog:%d, %d\n", frog.x, frog.y);
        }
```

After all the initialization, the main thread **begins its routine**. If not exiting the game, the main thread is responsible for **detecting keyboard input**. If typing 'q' or 'Q', it changes the status immediately and quits the game. If typing 'w' or 'W', it **updates** the frog and map.

```
else if (input == 's' || input == 'S'){
    if (frog.x == 10){
        // pthread_mutex_lock(&status_lock);
        status = 2;
        // pthread_mutex_unlock(&status_lock);
        break;
    }
    pthread_mutex_lock(&frog_lock);
    ++frog.x;
    pthread_mutex_unlock(&frog_lock);

    pthread_mutex_lock(&map_lock);
    map[frog.x][frog.y] = '0';
    map[frog.x-1][frog.y] = '=';
    pthread_mutex_unlock(&map_lock);

    // printf("after type in w, frog:%d, %d\n", frog.x, frog.y);
```

If typing 's' or 'S', first **detect** whether frog is on the **border**. If it is, lose game and quit. Otherwise, **update** the map and frog. When typing 'a', 'A', 'd' or 'D' the logic is similar.

```
for (i = 0; i < ROW+1; ++i){
    pthread_join(threads[i], NULL);
}
switch (status){
    case 1: printf("\033[2J\033[HYou win!!!!\n"); break;
    case 2: printf("\033[2J\033[HYou lose!!!\n"); break;
    case 3: printf("\033[2J\033[HYou quit the game.\n"); break;
}


// pthread_mutex_destroy(&status_lock);
pthread_mutex_destroy(&frog_lock);
pthread_mutex_destroy(&map_lock);
pthread_exit(NULL);
return 0;
```

When getting out of the while loop, it means **it's time to exit the game**. So recycle all child threads and print corresponding message according to the status.
Now let's go through the child thread.

```
void *logs_move( void *t ){
    int id = (intptr_t) t;
    int begin_index;
    int end_index;
    // printf("I'm thread %d\n", id); /* there are totally 11 new threads, from id = 0 to id = 10 */
    /*  initialize the logs  */
    if (id != 0 && id != 10){
        int initial_place = rand() % 49;
        for (int i = 0; i < LOGLEN; ++i){
            pthread_mutex_lock(&map_lock);
            map[id][(initial_place+i)%49] = '=';
            pthread_mutex_unlock(&map_lock);
        }
        begin_index = initial_place;
        end_index = (begin_index + 14)%49;
    }

    printf("\033[?25l");
    while(!status){
```

**For each line of wood, initialize it in the random place, that is to put the '='s from the random beginning index and the following 14 indexes**.

Then enter the routine loop.

```
while(!status){
    if (id == 10){
        printf("\033[%d;1H%s", id+1, map[10]);
        usleep(50000);
        continue;
    }

    else if (id == 0){
        printf("\033[%d;1H%s", id+1, map[0]);
        if (frog.x == id){
            status = 1;
            continue;
        }
        usleep(50000);
        continue;
    }
```

For special cases, when this thread is responsible for the first line or last line, no **losing status** detection. And for the first line, **success detection is needed**. If no status change, **sleep for 0.5s as the update rate period of game display.**

```c
else if (!(id%2)){ // in 2, 4, 6, 8 line, move right
    printf("\033[%d;1H%s", id+1, map[id]);
    char temp[COLUMN];
    for (int i = 0; i < COLUMN-1; ++i){
        temp[i] = map[id][(i-1+49)%49];
    }
    temp[COLUMN-1] = '\0';

    begin_index = (begin_index+1)%49;
    end_index = (begin_index + 14)%49;

    pthread_mutex_lock(&map_lock);
    strcpy(map[id], temp);
    pthread_mutex_unlock(&map_lock);

    if (frog.x == id){
        pthread_mutex_lock(&frog_lock);
        frog.y++;
        pthread_mutex_unlock(&frog_lock);

        if ((begin_index < end_index && (frog.y < begin_index || frog.y > end_index)) ||
            (begin_index > end_index && (frog.y > end_index && frog.y < begin_index)) ||
            (frog.y == 49)){
            status = 2;
            continue;
        }
    }
    usleep(50000);
}
```

For the even indexed lines, the wood will move right. To update the map, a `temp` is maintained to store the next state of this row of map. What's more, the **losing game detection is implemented here by detecting whether the frog is in not in the wood or the frog hits the border.** The `begin_index` and `end_index` are maintained for this detection. **If no status change, sleep for 0.5s as the update period of game display.**

```c
else if (id%2){ // in 1, 3, 5, 7, 9 line, move left
    printf("\033[%d;1H%s", id+1, map[id]);
    char temp[COLUMN];
    for (int i = 0; i < COLUMN-1; ++i){
        temp[i] = map[id][(i+1+49)%49];
    }
    temp[COLUMN-1] = '\0';

    begin_index = (begin_index-1+49)%49;
    end_index = (begin_index+14)%49;

    pthread_mutex_lock(&map_lock);
    strcpy(map[id], temp);
    pthread_mutex_unlock(&map_lock);

    if (frog.x == id){
        pthread_mutex_lock(&frog_lock);
        frog.y--;
        pthread_mutex_unlock(&frog_lock);

        if ((begin_index < end_index && (frog.y < begin_index || frog.y > end_index)) ||
            (begin_index > end_index && (frog.y > end_index && frog.y < begin_index)) ||
            (frog.y == -1)){
            status = 2;
            continue;
        }
    }
    usleep(50000);
}
```

For the odd indexed lines, the logic is similar to the case when line index is even number, except that the wood will move left.

```c
printf("\033[2J");
pthread_exit(NULL);
```

When quiting the loop, clear the screen and thread exit to join the main thread.

# 2.2 Bonus

## 2.2.1 async.h

```c
typedef struct thread_pool_task{
    void (*task_function)(int);
    int args;
    struct thread_pool_task* next;
} pool_task_t;

typedef struct thread_pool{
    int  max_thread;
    pthread_t* thread_ids;
    pool_task_t*  thread_pool_head;
    pthread_mutex_t queue_lock;
    pthread_cond_t  queue_ready;
} thread_pool_t;

void async_init(int);
void async_run(void (*fx)(int), int args);
```

In `async.h` , I claim two struct types and two functions.

1. The `pool_task_t` represents the **task** in the pool task queue. So, for an **abstract task**, it has a `task_function` to handle the task. Of course, the function needs the arguments to implement. Since the function `async_run(void (*fx)(int), int args)` returns `void` and accepts `int` argument, the `task_function` also returns void and accepts `int` argument. In the task queue, it only needs a pointer to refer to the next task with `next` attribute, since in the queue the **head task will be handled first** and **then the next task to be handled**.
2. The `thread_pool_t` is the thread pool type. It includes an integer attribute `max_thread` representing the **maximum number of threads in this pool to handle tasks**. The `thread_ids` attribute is a **pointer to the child created thread from all the child threads.** The `thread_pool_head` attribute is a **pointer to the task queue head**, so it is a `pool_task_t` type pointer. The attributes `queue_lock` and `queue_ready` are used for protecting the **shared task queue in the pool** and **waking up one child thread to handle the task**, as mentioned before.

## 2.2.2 async.c

```c
static thread_pool_t pool;
```

```c
void async_init(int num_threads) {
    pool.max_thread = num_threads;
    pool.thread_ids = (pthread_t*)malloc(sizeof(pthread_t)*num_threads);
    pool.thread_pool_head = NULL;

    if (pool.thread_ids == NULL){
        printf("ERROR: malloc thread idsfails.\n");
        exit(1);
    }
    if (pthread_mutex_init(&(pool.queue_lock), NULL) != 0){
        printf("ERROR: queue_lock init fails.\n");
        exit(1);
    }
    if (pthread_cond_init(&(pool.queue_ready), NULL) != 0){
        printf("ERROR: queue_ready init fails.\n");
        exit(1);
    }
    for (int i = 0; i < num_threads; ++i){
        if (pthread_create(&(pool.thread_ids[i]), NULL, routine_fun, NULL) != 0){
            printf("pthread_create fails!\n");
            exit(1);
        }
    }

    return;
}
```

The `async_init` function innitializes a static variable `thread_pool_t pool` and creates all the child threads. The code is really simple and no extra instruction is needed. The `routine_fun` is the child thread function, so let's go through this function next.

```c
void* routine_fun(void* args){
    pool_task_t* task;
    while(1){
        pthread_mutex_lock(&(pool.queue_lock));
        while (!pool.thread_pool_head){
            pthread_cond_wait(&(pool.queue_ready), &(pool.queue_lock));
        }
        task = pool.thread_pool_head;
        pool.thread_pool_head = pool.thread_pool_head->next;
        pthread_mutex_unlock(&(pool.queue_lock));
        task->task_function(task->args);
        free(task);
    }

    return NULL;
}
```

This routine function is the task handler child thread. Every thread is responsible for one task. Thread will acquire the queue **head task** if there is a new task enqued to `pthread_cond_signal` **or there is some tasks remaining in the queue already**, which means the thread will not enter the `while(!pool.thread_pool_head)` to sleep until the new task enqueues. **After waked up**, the thread will let head task points to **next task in order to let another thread to handle it**. Then this thread handles this task alone. After handling it, free the memory of this task and continue to wait for next task.

```c
void async_run(void (*handler)(int), int args) {
    pool_task_t* task = malloc(sizeof(pool_task_t));
    pool_task_t* queue_end;

    if (!handler){
        printf("ERROR: no handler function in task!\n");
        exit(1);
    }

    task->task_function = handler;
    task->args = args;
    task->next = NULL;

    pthread_mutex_lock(&(pool.queue_lock));
    queue_end = pool.thread_pool_head;
    if (queue_end == NULL){
        pool.thread_pool_head = task;
    }
    else{
        while(queue_end->next){
            queue_end = queue_end->next;
        }
        queue_end->next = task;
    }
    pthread_cond_signal(&(pool.queue_ready));
    pthread_mutex_unlock(&(pool.queue_lock));
    // handler(args);
    return;
}
```

`async_run` adds new task to the task queue end. As long as a new task is coming, **raise the signal to wake up** one child thread to the task queue head. And of course when **modifying the task queue**, **mutex lock should be locked for protection**. This part is simple, no extra instruction is needed.

# Part 3: How to Run My Code and Demo Output

## 3.1 Main Task

Under the `/Assignment_2_120090575/source` directory, type `make` to start the game.

A demo show is here:

```
vagrant@csc3150:~/csc3150/Assignment_2_120090575/source$ make
```

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

**win the game**:

```
You win!!!!
vagrant@csc3150:~/csc3150/Assignment_2_120090575/source$
```

**lose the game**:

```
You lose!!!
vagrant@csc3150:~/csc3150/Assignment_2_120090575/source$
```

**quit the game**:

```
You quit the game.
vagrant@csc3150:~/csc3150/Assignment_2_120090575/source$
```

# 3.2 Bonus

1. type `make` under the `Assignment_2_120090575/3150-p2-bonus-main/thread_poll/httpserver` directory.

```
vagrant@csc3150:~/csc3150/Assignment_2_120090575/3150-p2-bonus-main/thread_poll$
 make
gcc -ggdb3 -c -Wall -std=gnu99 httpserver.c -o httpserver.o
gcc -ggdb3 -c -Wall -std=gnu99 libhttp.c -o libhttp.o
gcc -ggdb3 -c -Wall -std=gnu99 util.c -o util.o
gcc -ggdb3 -c -Wall -std=gnu99 async.c -o async.o
gcc -pthread httpserver.o libhttp.o util.o async.o -o httpserver
```

2. In this terminal, type `./httpserver --files files/ --port 8000 --num-threads 10` .

```
vagrant@csc3150:~/csc3150/Assignment_2_120090575/3150-p2-bonus-main/thread_poll$
 ./httpserver --files files/ --port 8000 --num-threads 10
Listening on port 8000...
```

3. In **another** terminal, type `ab -n 5000 -c 10 http://localhost:8000/` .

```
Completed 5000 requests
Finished 5000 requests


Server Software:
Server Hostname:        localhost
Server Port:            8000

Document Path:          /
Document Length:        4626 bytes

Concurrency Level:      10
Time taken for tests:   0.699 seconds
Complete requests:      5000
Failed requests:        0
Total transferred:      23460000 bytes
HTML transferred:       23130000 bytes
Requests per second:    7151.03 [#/sec] (mean)
Time per request:       1.398 [ms] (mean)
Time per request:       0.140 [ms] (mean, across all concurrent requests)
Transfer rate:          32766.24 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.1      0       2
Processing:     0    1  10.8      0     223
Waiting:        0    1  10.8      0     223
Total:          0    1  10.8      1     223

Percentage of the requests served within a certain time (ms)
  50%      1
  66%      1
  75%      1
  80%      1
  90%      1
  95%      1
  98%      1
  99%     34
 100%    223 (longest request)
```

This means the program successfully uses 10 threads to handle 5000 http requests within a limited time.

# Part 4: Environment

Distributor ID: Ubuntu Description: Ubuntu 16.04.7 LTS Release: 16.04 Codename: xenial
kernel version: 5.10.146

# Part 5: What Have I Learned from this Assignment

1. About the multithreaded frog game, I firstly tried to achieve the game using a **serial and non-parallel logic**, but only finding that the game had **a little bit latency**, which did not meet the requirement of a game. Nevertheless, by using multi-thread design, the game has **low latency that cannot be sensed by human**. What's more, the logic to achieve the game is much easier. Every thread is **responsible for its own thing**. When the shared resource is accessed or modified, protection is needed.
2. As for the bonus task, first thing I learnt is what a **thread pool** is. It's just about a number of threads created first, then they wait for the task queue to come in and handle this task queue. No thread will exit and be newly created after pool initialized. This will **reduce the cost of creating and destroying thread**.