

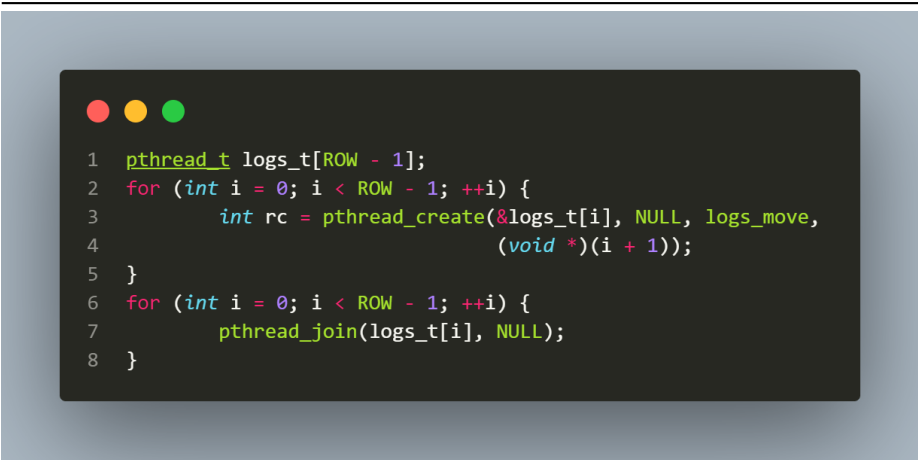
# Multi-Thread Programming

Yohandi [SID: 120040025]

## Program Design

In this task, we were instructed to create a minigame, namely Frog Crossing River, in which a frog must cross a river by leaping on moving logs. There are nine logs that move alternately to the left or right depends on its row position. Thus, the designed software will utilize nine threads, each of which will be accountable for a moving log. The program comprises primarily of one function kbhit for receiving keyboard input and two substantial routines, main and logs\_move.

The main function is responsible for initiating all subsequent functions. Three main aspects are programmed in the main function. Those are map fittings, threads creation, and output display. In the game map, it is noticable that both row 0 and 10 are filled with 50 | that are set as the start and finish area for the frog. This code part already exists in the code template. For the threads creation, we will use multithreading method. The reason is simply because threads are popular way to improve application through parallelism and they operate faster than processes. To create the threads, we use `pthread_create` function that exists within the POSIX Threads library. Additionally, to join the threads, we use `pthread_join` function. For the output display, we use a variable named `exit_status` to determine whether the game is stil on going, finished (with a lose / win result), or quitted. Then, we can print the relevant message accordingly.

A screenshot of a code editor with a dark background and light-colored text. The code is in C and shows the creation and joining of threads. It includes comments in Chinese. The code is as follows:

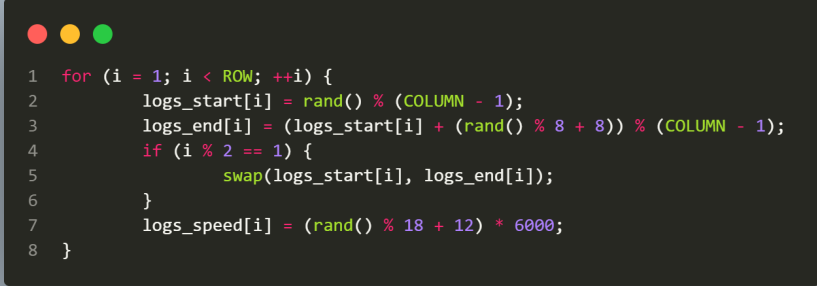
```
1 pthread_t logs_t[ROW - 1];
2 for (int i = 0; i < ROW - 1; ++i) {
3     int rc = pthread_create(&logs_t[i], NULL, logs_move,
4                             (void *) (i + 1));
5 }
6 for (int i = 0; i < ROW - 1; ++i) {
7     pthread_join(logs_t[i], NULL);
8 }
```

creating thread

For each log, some variables are served as properties that will enhance the

randomization aspect in the game mechanism. In its implementation, there are `logs_start` that determines the starting position of the current log, `logs_end` that determines the ending position of the current log, and `logs_speed` that determines the speed of the current log. In this case the higher the value of `logs_speed` the slower the log will move.

---



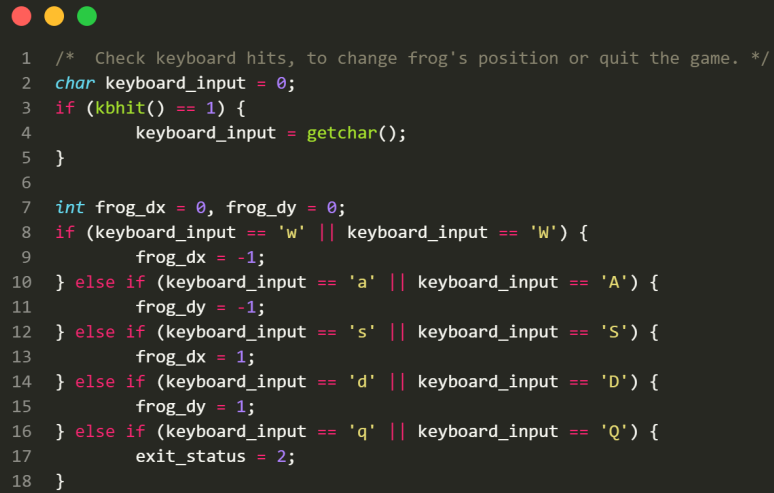
```
1 for (i = 1; i < ROW; ++i) {
2     logs_start[i] = rand() % (COLUMN - 1);
3     logs_end[i] = (logs_start[i] + (rand() % 8 + 8)) % (COLUMN - 1);
4     if (i % 2 == 1) {
5         swap(logs_start[i], logs_end[i]);
6     }
7     logs_speed[i] = (rand() % 18 + 12) * 6000;
8 }
```

---

logs variables

---

In `logs_move` function, however, four main aspects are programmed. Those are logs move, keyboard buttons hit, game status determine, and map print. While moving the logs, we only need to consider the oddity of the current row. The odd rows are specified to move from right to the left. Similarly, the even rows move from left to the right. After that, we simply add / reduce 1 point from the start and end of the log for the current row. For the keyboard hit, we utilize the use of `kbhit` function to determine whether user is hitting a keyboard button. If the function returns true, we use `getchar` to get the desired input. Then, depending on the move input, we deduce the value of `frog_dx` and `frog_dy` accordingly. Both variables determine the delta value of the x and y position for the frog. In a game status, however, we first clarify the value -1 as a signal that implies the game is still on going. Other than that, we use value from 0 to 2 to determine the finish status. Lastly, to print the map we simply create all the fittings that are similar to the map in the main function. Then, to display the logs, we simply loop over from start to end along with the direction which is described by the oddity value of the current row. For the frog, however, we also need to handle a case where it hits the finish area, it collides with the boundary, etc.



```
1  /* Check keyboard hits, to change frog's position or quit the game. */
2  char keyboard_input = 0;
3  if (kbhit() == 1) {
4      keyboard_input = getchar();
5  }
6
7  int frog_dx = 0, frog_dy = 0;
8  if (keyboard_input == 'w' || keyboard_input == 'W') {
9      frog_dx = -1;
10 } else if (keyboard_input == 'a' || keyboard_input == 'A') {
11     frog_dy = -1;
12 } else if (keyboard_input == 's' || keyboard_input == 'S') {
13     frog_dx = 1;
14 } else if (keyboard_input == 'd' || keyboard_input == 'D') {
15     frog_dy = 1;
16 } else if (keyboard_input == 'q' || keyboard_input == 'Q') {
17     exit_status = 2;
18 }
```

hitting keyboard

```

1  for (int j = 0; j < COLUMN - 1; ++j) {
2      map[0][j] = '|';
3      map[ROW][j] = '|';
4  }
5  for (int j = shift(logs_end[cur_row], dx[cur_row % 2]);
6      j != logs_start[cur_row]; j = shift(j, dx[cur_row % 2])) {
7      map[cur_row][j] = ' ';
8  }
9  for (int j = logs_start[cur_row];
10     j != shift(logs_end[cur_row], dx[cur_row % 2]);
11     j = shift(j, dx[cur_row % 2])) {
12     map[cur_row][j] = '=';
13 }
14 map[frog.x][frog.y] = (char)('0');
15
16 /* Print the map on the screen */
17 printf("\033[2J\033[H");
18 for (int i = 0; i <= ROW; ++i) {
19     puts(map[i]);
20 }

```

---

printing map

---

## Bonus

According to the reference given, a thread pool is a software design technique that enables concurrent program execution. By maintaining a pool of threads, the approach boosts performance and eliminates execution delays caused by the frequent creation and destruction of threads for short-lived processes. The number of active threads is proportional to the program's available computing resources, such as a parallel task queue once program execution is complete.

In this bonus task, we are asked to implement both `async_init` and `async_run` function in `async.h` and `async.c` files. The implementation requires us to make both function supports the application of thread pool in the server.

---



```
1 void async_init(int num_threads) {
2     /** TODO: create num_threads threads and initialize the thread pool **/
3     if(num_threads > 0) {
4         pthread_t *threads = (pthread_t *) malloc(sizeof(pthread_t)*num_threads);
5         pthread_t *p = threads;
6         my_init(&queue_work);
7
8         // create num_threads threads + thread pool
9         for(int i = 0; i < num_threads; i++, p++) {
10             pthread_create(p, NULL, thread_pool, NULL);
11         }
12     }
13
14     return;
15 }
```

---

initializing async

---

In its implementation, `async_init` requires a queue to hold all the thread requests. Therefore, we also need to implement the queue accordingly.

```

1 // perform a casual "init" action from the queue (initialize)
2 void my_init(my_queue_t *my_queue){
3     my_queue -> head = NULL;
4     my_queue -> size = 0;
5 }
6 // perform a casual "push" action from the queue (add item)
7 void my_push(my_queue_t *my_queue, int client_socket_fd){
8     my_item_t *my_item = calloc(1, sizeof(my_item_t));
9     my_item -> client_socket_fd = client_socket_fd;
10
11     pthread_mutex_lock(&mutex);
12
13     DL_APPEND(my_queue -> head, my_item);
14     my_queue -> size++;
15
16     pthread_cond_signal(&cond);
17     pthread_mutex_unlock(&mutex);
18 }
19 // perform a casual "pop" action from the queue (remove item)
20 int my_pop(my_queue_t *my_queue){
21     pthread_mutex_lock(&mutex);
22
23     while(!my_queue -> size){
24         pthread_cond_wait(&cond, &mutex);
25     }
26     my_item_t *my_item = my_queue -> head;
27     int client_socket_fd = my_queue -> head -> client_socket_fd;
28     my_queue -> size--;
29     DL_DELETE(my_queue -> head, my_queue -> head);
30     pthread_mutex_unlock(&mutex);
31
32     free(my_item);
33     return client_socket_fd;
34 }
35

```

queue implementation

## Program Execution

### Program Environment

- Linux Kernel Version: Linux-5.10.146
- GCC Version: gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.12)

To set up the development environment, we first install both Virtual Machine (from the virtualbox official website) and Vagrant (from the vagrantup official website). Then, we launch PowerShell as an administrator to up a vagrant for the CSC3150 folder with the main server, cyzhu/csc3150. Then, we set up the ssh remote in Visual Studio Code by installing the Remote - SSH extension and config it with the SSH-TARGETS. After copying all the necessary information, we will be able to open a new window named default. Last, we open a new terminal and install all essential dependencies and libraries by typing `sudo apt update && sudo apt install -y build-essential`.

### Execution Steps

- Open a new terminal and type `cd {path}` to enter the directory of the program.
- Type `g++ hw2.cpp -lpthread` to compile the program.
- Type `./a.out` to execute the program.

For the bonus program: - Open a new terminal and type `cd {path}` to enter the directory of the program. - Type `make` to run a makefile and compile all programs. - Type `./httpserver --files files/ --port 8000 --num-threads {number of threads}` and replace the number of threads with an integer number to execute the program.

### Result



The screenshot shows a terminal window with a progress bar at the top consisting of vertical bars and a single '0' in the middle. Below the progress bar, there are several lines of text, including 'win' and a command prompt 'vagrant@csc3150:~/csc3150/AS2\_120040025/source\$'.

win verdict

[illegible]

---

lose verdict

---

```

|||||
=====

=====
=====

=====

=====
=====0==

=====
=====

=====

|||||
quit
vagrant@csc3150:~/csc3150/AS2_120040025/source$ █

```

---

quit verdict

## Reflection

I attempted to create the game without utilizing pthread when I first began working on this assignment. The resulting game is rather lagging, especially



when the user provides input. After completing the game using pthread, I compare the results and determine that the performance of the pthread version is significantly superior to that of the non-pthread version. As a result, I have learned that utilizing pthread for this type of application can improve its efficiency because the weight of program execution is dispersed across multiple threads. In addition, since the speed of a computer is significantly greater than that of a human, utilizing a thread that is dedicated to receiving user input can ensure that the slow reaction of humans does not impair the overall performance of the program. In addition, I found it both entertaining and demanding to learn how to perform multi-threading using the c++ programming language in a Linux environment. It enables me to run many threads to execute a section of the program's code so that the program can run concurrently, thereby considerably enhancing the execution speed.