

Report

Yifu TIAN
121090517

BONUS

Task Retell

Bonus requires to implement two key functions of a thread pool:

- `async_init(int num_threads)` : This function should create ‘num_threads’ threads and initialize the thread pool.
- `async_run(void (*handler)(int), int args)`: This function should be modified to support the thread pool and should run tasks asynchronously without directly invoking `pthread_create`.
- We can utilize list data structures from `utlist.h`.
- When there's no work, the threads created by `async_init` should sleep, but they must wake up immediately when a job arrives, avoiding busy waiting.
- The given `async_run` implementation runs tasks synchronously, and we need to rewrite it to be asynchronous.

Design and Implementation

A. For `thread_function`:

This function represents the main logic that each thread in the thread pool will execute. Firstly, each thread increments a global ‘thread_num’ counter to identify itself. Then the thread enters an infinite loop where it will wait until there are tasks available in the thread pool queue ‘thread_pool’. When a task is available, the thread removes it from the queue, unlocks the mutex, processes the task and then frees the

memory used by the task.

B. For `async_init` function:

This function is responsible for initializing the thread pool and its associated synchronization primitives. A queue (`thread_pool`) is allocated in memory to store tasks. Mutex (`mutexQueue`) and condition variable (`condQueue`) are initialized to synchronize access to the task queue. A fixed number of threads (specified by `num_threads`) are created. Each thread runs the `thread_function`.

C. For `async_run` function:

This function allows users to submit tasks (functions and their arguments) to the thread pool for asynchronous execution. A new task (`work_items`) is allocated in memory and initialized with the provided function (`handler`) and arguments (`args`). The task is then added to the end of the task queue (`thread_pool`), and the size of the queue is incremented. The condition variable (`condQueue`) is signaled to wake up a waiting thread (if any) to process the newly added task.

Execution

1. Enter 'make' in the terminal under the root folder of `async.c`
2. Enter '`./httpserver --proxy inst.eecs.berkeley.edu:80 --port 8000 --num-threads 5`'
3. When you run the test, you can access `127.0.0.1:8000` at your browser even before modifying the code, but it cannot serve the request concurrently (multiple requests at the same time) and always serve with the same thread id. After implementing the thread pool you should support concurrent access.

Homework 2

Task Retell

In homework2, we need to implement the multithreaded game "Frog crossed river". The game rules and details are given as follows:

1. A river contains logs floating on it, and a frog aims to cross the river by

jumping on these logs.

2. The objective of the game is to get the frog from one bank of the river to the other successfully. Players win if the frog jumps to the opposite bank of the river successfully; Players lose if the frog lands in the river or if a log reaches the edge of the screen with the frog still on it.

3. The frog starts in the middle of the bottom river bank and players can control the frog's jumps using the keyboard. Logs can move staggeredly from left to right or vice versa.

4. Function Requirements:

A. Complete the function "logs_move" to allow staggered movement of logs.

B. Utilize pthread and mutex locks for controlling the movement of logs and the frog.

C. Use the provided "kbhit" function for keyboard input capture and update the frog's position based on these inputs.

D. The program should determine the game's status (win, lose, or quit) and display appropriate messages to the user.

Environment

- Linux Version

```
● vagrant@csc3150:~/csc3150/Assignment_2_121090517$ cat /etc/issue
Ubuntu 16.04.7 LTS \n \l
```

- Linux Kernel Version

```
● vagrant@csc3150:~/csc3150/Assignment_2_121090517$ uname -r
5.10.196
```

- GCC Version

```
● vagrant@csc3150:~/csc3150/Assignment_2_121090517$ 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.
```

Design and Implementation

For main function, it includes mainly four parts:

A. Initialize the river map and frog's starting position;

```
308  int main( int argc, char *argv[] ){
309      // Initialize the river map and frog's starting position
310      memset( map , 0, sizeof( map ) ) ;
311      int i , j ;
312      for( i = 1; i < ROW; ++i ){
313          for( j = 0; j < COLUMN - 1; ++j )
314              map[i][j] = ' ' ;
315      }
316      for( j = 0; j < COLUMN - 1; ++j )
317          map[ROW][j] = map[0][j] = '|' ;
318
319      for( j = 0; j < COLUMN - 1; ++j )
320          map[0][j] = map[0][j] = '|' ;
321
322      frog = Node( ROW, (COLUMN-1) / 2 ) ;
323      map[frog.x][frog.y] = '0' ;
324      /*****
```

B. Initialize the logs by call the function initLogs() and print the map into screen;

```
324      /*****
325      initLogs();
326      //Print the map into screen
327      for( i = 0; i <= ROW; ++i)
328          puts( map[i] );
```

```

45 void initLogs(){
46     // initialize log positions and lengths
47     srand(time(NULL));
48
49     LOGPOS = (int*)malloc(sizeof(int) * (ROW-1));
50     LOGLENGTH_ARRAY = (int*)malloc(sizeof(int) * (ROW-1));
51     for (int i=0; i<ROW-1; i++) {
52         LOGPOS[i] = rand() % (COLUMN-1);
53         LOGLENGTH_ARRAY[i] = LOGLENGTH + rand() % LOGLENGTHVAR;
54         int start = LOGPOS[i];
55         int len = LOGLENGTH_ARRAY[i];
56         for (int k=0; k<len; k++) {
57             map[i+1][(start+k+(COLUMN-1))%(COLUMN-1)] = '=';
58         }
59     }
60     printf("\033[H\033[2J");
61 }

```

The function 'initLogs()' initializes the positions and lengths of logs. Firstly, set a random seed based on the current time. Then allocates memory for storing the positions and lengths of logs. Each log's position is randomly determined based on the number of columns, and its length is also randomly determined with variability. Next, update the game map to represent the logs' positions and length. Finally, print out an escape sequence 'printf("\033[H\033[2J")' to clear the terminal screen and move the cursor position.

C. Create pthreads for log move, frog control and call functions to control game status, update game map.

```

330     getUserInput();
331
332     pthread_mutex_init(&map_mtx, NULL);
333     pthread_mutex_init(&game_mtx, NULL);
334     long k;
335     /* Create pthreads for wood move and frog control. */
336     pthread_t threads[ROW+1];
337
338     // Create log threads
339     for (k = 0; k < ROW-1; k++) {
340         createThread(&threads[k], logs_move, (void*)k);
341     }
342     // Create frog move control and game status control threads
343     createThread(&threads[ROW-1], frog_move, NULL);
344     createThread(&threads[ROW], game_control, NULL);
345     // Wait for threads to finish
346     for (k = 0; k < ROW-1; k++) {
347         joinThread(threads[k]);
348     }
349     joinThread(threads[ROW-1]);
350     joinThread(threads[ROW]);
351     /* Update game map based on user's game status: win, lose, or quit. */
352     if (!BOUND) {
353         printUpdatedMap();
354         usleep(3*SPEED);
355     }
356     printf("\033[?25h\033[H\033[2J");

```

Next, initialize mutexes for game synchronization. Create multiple threads to manage the movement of logs and the control of frog in the game. Then wait for these threads to finish their tasks. After the threads complete, the game map will be updated based on the player's game status(win, lose, or quit). The map is then displayed, followed by a brief pause and a terminal command to hide the cursor for visual clarity.

Here I will give more demonstration about the functions I created.

I) createThread & joinThread

```

288 // Define thread function types
289 typedef void *(*ThreadFunction)(void *);
290 // Utility function to create threads
291 void createThread(pthread_t *thread, ThreadFunction function, void *args) {
292     if (pthread_create(thread, NULL, function, args) != 0) {
293         perror("Thread creation failed");
294     }
295 }

```

```

296 // Utility function to join threads
297 void joinThread(pthread_t thread) {
298     if (pthread_join(thread, NULL) != 0) {
299         perror("Thread join failed");
300     }
301 }

```

I define a type for thread functions and provides a utility function to create threads. The function 'createThread' attempts to create a new thread and prints out error message if thread creation fails. In the same way, function 'joinThread' attempts to join a specified thread and prints and error message if the joining process fails.

II) printUpdatedMap & printMap

```
202 void printMap(char tmpMap[ROW+10][COLUMN], Node curFrogPos){
203
204     // Copy the current state of the map to a temporary map
205     cloneMapState(tmpMap);
206
207     // Get the current position of the frog
208     getCurrentFrogPosition(&curFrogPos);
209
210     // Update the temporary map to show the river banks and the frog
211     for(int j = 0; j < COLUMN - 1; ++j) {
212         tmpMap[ROW][j] = map[0][j] = '|';
213         tmpMap[0][j] = map[0][j] = '|'; // Draw the banks
214     }
215     tmpMap[curFrogPos.x][curFrogPos.y] = '0'; // Draw the frog
216
217     // Print the updated map
218     printf("\033[H\033[2J"); // Clear the console
219     for (int i = 0; i < ROW + 1; i++) {
220         puts(tmpMap[i]);
221     }
222 }
223 void printUpdatedMap() {
224     char tmpMap[ROW + 10][COLUMN];
225     Node curFrogPos;
226     printMap(tmpMap, curFrogPos);
227     getUserInput();
228 }
```

For 'printMap' function, it takes in a temporary map 'tmp' and the current position of the frog 'curFrogPos'. It copies the current game map status to the temporary map and retrieves the current frog position. Then update the temporary map to illustrate the river banks and places the frog on its current position. Finally clear the console screen and then prints the updated map line by line.

For 'printUpdatedMap' function, I declare a temporary map and a variable for the frog's position. Then call the 'printMap' function to draw and display the game map with the frog's position. Afterwards, get user input for the next move or action.

D. Print out the game status and call the function cleanup() to destroy

map mutex, game mutex and free the memory I created. Finally use `pthread_exit()` to exit.

```
357     /* Display the output for user: win, lose or quit. */
358     if (GAMESTATUS <= EXIT) {
359         printf("%s\n", messages[GAMESTATUS]);
360     }
361     /*******/
362     cleanup();
363     return 0;
364 }
```

```
302 // clean up
303 void cleanup() {
304     pthread_mutex_destroy(&map_mtx);
305     pthread_mutex_destroy(&game_mtx);
306     free(LOGPOS);
307     pthread_exit(NULL);
308 }
```

Here I create a messages array to enhance the program scalability. Display the output for user and clean up all the mutex I have created. Finally call 'pthread_exit' to exit.

```
29     const char *messages[4] = {
30         NULL,
31         "You win the game!",
32         "You lose the game!",
33         "You quit the game!"
34     };
```

For other functions, here are something important to mention:

A. Game_control function

```
259 void *game_control(void *t){
260     while (isGamePlaying()){
261         printUpdatedMap();
262         usleep(SPEED);
263     }
264     pthread_exit(NULL);
265 }
```

This function serves as the main game loop for updating and rendering the game state at a specific pace(set by constant 'SPEED'). It continually checks if the game is

still playing using the 'isGamePlaying()' function. Once the game is no longer in progress, the loop exits and the function ends the thread using 'pthread_exit(NULL)'. This structure helps ensure that the game's state is regularly updated and displayed to the player, creating a dynamic and real-time gaming experience.

B. Frog_move function

```
229 void *frog_move(void *t){
230     while(isGamePlaying()){
231         if (kbhit()){
232             char move_direction = getchar();
233             switch(move_direction){
234                 case 'w':
235                 case 'W':
236                     check_and_move_frog(-1, 0);
237                     break;
238                 case 's':
239                 case 'S':
240                     check_and_move_frog(1, 0);
241                     break;
242                 case 'a':
243                 case 'A':
244                     check_and_move_frog(0, -1);
245                     break;
246                 case 'd':
247                 case 'D':
248                     check_and_move_frog(0, 1);
249                     break;
250                 case 'q':
251                 case 'Q':
252                     changeStatus(EXIT);
253             }
254         }
255         updateGameStatus();
256     }
257     pthread_exit(NULL);
258 }
```

This function is the logic responsible for controlling the movement of the frog in the game. It will continuously check if the game is still playing with the 'isGamePlaying()' function. If the game is in progress and a key is pressed by player (checked by 'kbhit()' function), it fetches the key input using 'getchar()'. Depending on the character input, the function makes a decision:

- 'w' or 'W': Move the frog upwards by calling check_and_move_frog(-1, 0).
- 's' or 'S': Move the frog downwards by calling check_and_move_frog(1, 0).
- 'a' or 'A': Move the frog to the left by calling check_and_move_frog(0, -1).
- 'd' or 'D': Move the frog to the right by calling check_and_move_frog(0, 1).
- 'q' or 'Q': End the game by calling changeStatus(EXIT).

After handling the input, the function updates the game status with 'updateGameStatus()'. When the game loop ends, the threads exits using 'pthread_exit(NULL)'.

C. Logs_move function

```
274 void *logs_move(void *t){
275     long id = (long)t;
276     int currentPos = LOGPOS[id];
277     int logLength = LOGLENGTH_ARRAY[id];
278     Node frogState;
279     while (isGamePlaying()) {
280         int moveDirection = (id % 2) ? 1 : -1;
281         currentPos = updatePosition(currentPos, moveDirection);
282         copyFrogData(&frogState);
283         changeMap_log(id, currentPos, logLength, frogState.x, frogState.y);
284         usleep(SPEED);
285     }
286     pthread_exit(NULL);
287 }
```

The function begins by converting the passed-in thread argument 't' to a long type to get the thread ID. Then initialize the log's current position (currentPos) and its length (logLength) using arrays LOGPOS and LOGLENGTH_ARRAY indexed by the log ID.

The while loop checks if the game is still ongoing using the 'isGamePlaying()' function.

Within this loop:

- The move direction for the log is determined based on whether the log ID is even or odd. Even IDs move in one direction (1) and odd IDs move in the opposite direction (-1).
- The 'updatePosition' function updates the current position of the log based on the move direction.
- The 'copyFrogData' function is used to fetch the current state or position of the frog.
- The 'changeMap_log' function presumably updates the game map to reflect the log's movement and possibly interactions with the frog.
- The function then sleeps for a duration defined by the 'SPEED' constant

before the next iteration.

After the loop (i.e., when the game is no longer playing), the function exits the thread using 'pthread_exit(NULL)'.

Compile & Execution

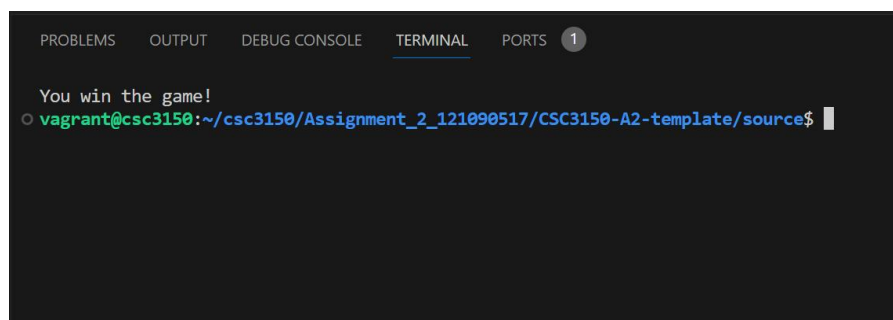
- HOW TO COMPILE: In the 'source' directory, type 'g++ hw2.cpp -lpthread' and enter on console.
- HOW TO EXECUTE: In the 'source' directory, type './a.out'

Screenshot of Output

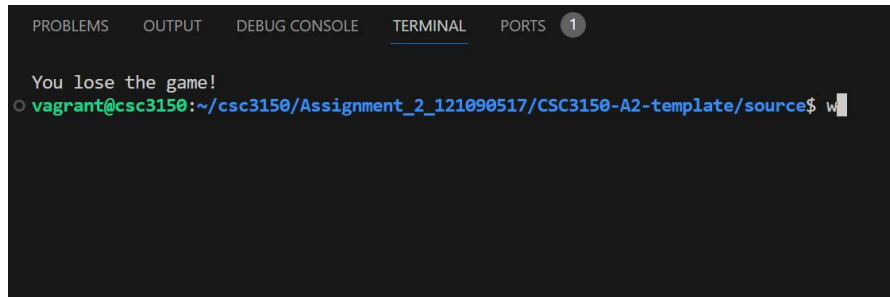
A. The Initial Interface



B. You Win the Game

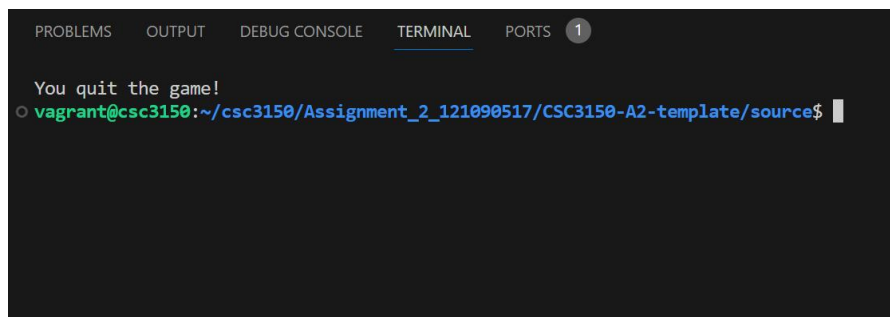


C. You Lose the Game



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS 1
You lose the game!
vagrant@csc3150:~/csc3150/Assignment_2_121090517/CSC3150-A2-template/source$ w
```

D. You Quit the Game



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS 1
You quit the game!
vagrant@csc3150:~/csc3150/Assignment_2_121090517/CSC3150-A2-template/source$
```

Learn from Project 1

From bonus, I learned:

- How to synchronize threads using mutexes (`pthread_mutex_t`) and condition variables (`pthread_cond_t`).
- The concept of thread pools and their utility in managing concurrent tasks efficiently.
- Memory management techniques, especially when working with dynamically allocated data structures in a multithreaded environment.
- The intricacies of task scheduling and execution in a thread pool setup.
- The significance of error handling, particularly in a concurrent programming context, to ensure the stability and robustness of the application

From homework2, I have gained several insights:

- Multi-threading: I learned about the intricacies of implementing multi-threaded programs, particularly using the `pthread` library. This includes creating, managing, and synchronizing threads, and understanding potential

challenges such as race conditions.

- **Game Logic Design:** The assignment provided an understanding of how game logic is structured, especially in the context of a simple game environment. This encompasses things like player inputs, game state updates, and rendering graphics on the console.
- **Memory Management:** In C++, proper memory management is crucial. This assignment reinforced the importance of allocating and deallocating memory appropriately to prevent memory leaks or other unexpected behaviors.
- **Problem Solving:** Encountering and resolving bugs or unexpected behaviors sharpened my problem-solving and debugging skills. It also emphasized the value of careful code review and testing.