The Chinese University of Hong Kong, Shenzhen

CSC3150

Operating System

Assignment 2 Report

Author: Yuzhe Yang

Student ID: 121090684

October 25, 2023

Contents

1 Environment 2

2 Design & Implementation 2

2.1 Overview 2

2.2 Logs Move (`logs_move` function) 3

2.3 Frog Move (`frog_move` function) 4

2.4 Mutexes (`log_mutex` and `frog_mutex`) 4

3 Program Result 4

4 Bonus 5

4.1 Environment 5

4.2 Overview 6

4.3 Design & Implementation 6

4.4 Program Result 7

5 What did I learn 7

1 Environment

`cat /etc/os-release` output:

```
PRETTY_NAME="Debian GNU/Linux 11 (bullseye)"
NAME="Debian GNU/Linux"
VERSION_ID="11"
VERSION="11 (bullseye)"
VERSION_CODENAME=bullseye
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"
```

`uname -a` output:

```
Linux msk-bullseye 5.10.191
```

`gcc --version` output:

```
gcc (Debian 10.2.1-6) 10.2.1 20210110
```

2 Design & Implementation

2.1 Overview

The Frogger game is designed with a modular and well-structured approach, incorporating multi-threading to handle concurrent tasks. The goal of the game is making the frog jump to the opposite riverside and avoid to drop into the water. The game is composed of two main threads: one responsible for moving logs horizontally (`logs_move`), and the other for controlling the frog's movement (`frog_move`). To ensure thread safety and prevent race conditions, mutexes (`log_mutex` and `frog_mutex`) are employed.

The program represents the game environment using a 2D array (`map`) where the positions of the frog, logs, and river elements are stored. User input is detected to control the frog's movements, and the main loop continuously updates and prints the game state.

The `game_status` variable serves as a crucial component in controlling the main game loop. It is a boolean flag that, when `true`, allows the game to continue. This variable is checked in both the `logs_move` and `frog_move` threads to determine when to exit and end the game. If the `game_status` is `false` which means the frog will encounter the water, the threads will exit, and the game will end.

The `alive_map` is a crucial 2D boolean array that signifies the "alive" positions on the game map. Each element `alive_map[i][j]` is `true` if there is an obstacle at position (i, j), and `false` if the position is empty.

This information is vital for controlling the frog's movement. The frog needs to navigate through the game map while avoiding moving obstacles such as vehicles or floating logs. By keeping track of these positions in `alive_map`, the game can determine whether the frog collides with obstacles and, consequently, whether the frog is alive.

The main function contains a loop that continually updates and prints the game state. In each iteration, the terminal is cleared for a smoother display. The loop checks for various game-ending conditions, such as winning, losing, or quitting, and updates the game status accordingly. The whole prcession of the program is displayed as Figure 1 below.

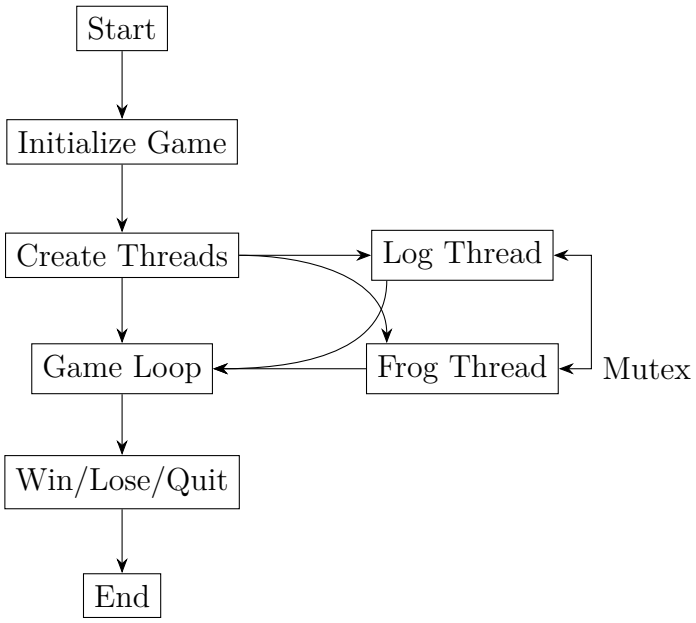


Figure 1: Program Framework of the Game

2.2 Logs Move (`logs_move` function)

The `logs_move` function is designed to run in a separate thread, simulating the movement of logs horizontally across the river. To ensure thread safety and prevent conflicts with the frog's movement, the function utilizes a mutex (`log_mutex`). In a loop, the positions of the logs are updated, and the `usleep` function introduces a delay, controlling the speed of log movement.

```
1 void *logs_move(void *t) {
2     while (game_status) {
3         pthread_mutex_lock(&log_mutex);
```

```
        // Code for updating log positions
        pthread_mutex_unlock(&log_mutex);
        usleep(SLEEP);
    }
    return NULL;
}
```

2.3 Frog Move (`frog_move` function)

The `frog_move` function, executed in its own thread, is responsible for handling user input to control the frog's movement. To synchronize access to shared data and avoid conflicts with the log-moving thread, the function employs a mutex (`frog_mutex`). User input, detected using the `kbhit` function, determines the frog's position updates. The function continually checks the game status, including whether the frog reaches the top, collides with obstacles, or if the player decides to quit.

```
void *frog_move(void *t) {
    while (game_status) {
        pthread_mutex_lock(&frog_mutex);
        // Code for handling user input and updating frog position
        pthread_mutex_unlock(&frog_mutex);
        usleep(SLEEP);
    }
    return NULL;
}
```

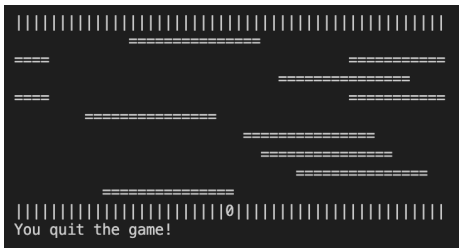
2.4 Mutexes (`log_mutex` and `frog_mutex`)

Mutexes are crucial for ensuring thread safety and preventing data race conditions. The `logs_move` and `frog_move` functions use the `log_mutex` and `frog_mutex` respectively to control access to shared data. This guarantees that only one thread at a time can modify critical game data, avoiding conflicts between the two threads.

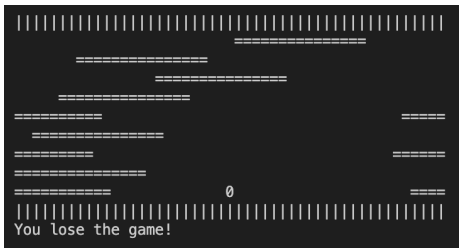
3 Program Result

The program outputs a dynamic representation of the game state in the console. The river, frog, and logs are visually represented, and the game status is continually updated. The terminal is cleared in each iteration for a smoother display. The game ends when the frog either reaches the top, collides with obstacles, or the player quits. **HOW TO COMPILE:** In the `source` directory, type `g++ hw2.cpp -o hw2 -lpthread` and press Enter in the console.

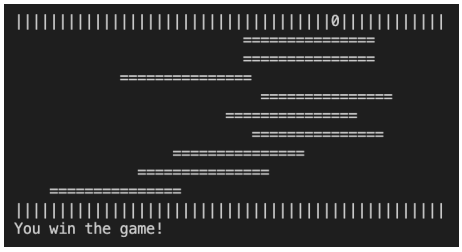
HOW TO EXECUTE: In the `source` directory, type `./hw2` and press Enter in the console.



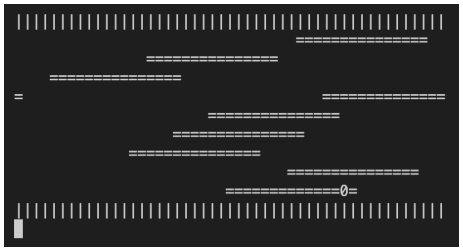
(a) Case 1: Quit



(b) Case 2: Lose



(c) Case 3: Win



(d) When the frog is on the log

Figure 2: Program Output

4 Bonus

4.1 Environment

I switched to the Ubuntu 22.04 system because it has a GUI interface, making it convenient for me to directly access web pages using the browser. I also test the program on the preview environment.

`cat /etc/os-release` output:

```
PRETTY_NAME="Ubuntu 22.04.2 LTS"
NAME="Ubuntu"
VERSION_ID="22.04"
VERSION="22.04.2 LTS (Jammy Jellyfish)"
VERSION_CODENAME=jammy
ID=ubuntu
ID_LIKE=debian
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"
UBUNTU_CODENAME=jammy
```

`uname -a` output:

```
Linux ubuntu-linux-22-04-02-desktop 5.15.0-76-generic
```

`gcc --version` output:

```
gcc (Ubuntu 11.3.0-1ubuntu1~22.04.1) 11.3.0
```

4.2 Overview

The program appears to be designed to handle asynchronous job processing. The key components of the system include a job queue and a set of worker threads. The queue holds jobs that are waiting to be processed, and the worker threads continuously check the queue to process any available jobs.

The program involves a globally shared queue, `job_queue`, where jobs are stored to be executed by worker threads. A worker thread is a standalone executable entity that waits for jobs in the queue, retrieves them, and executes them. The framework of the program is shown in Figure 3.

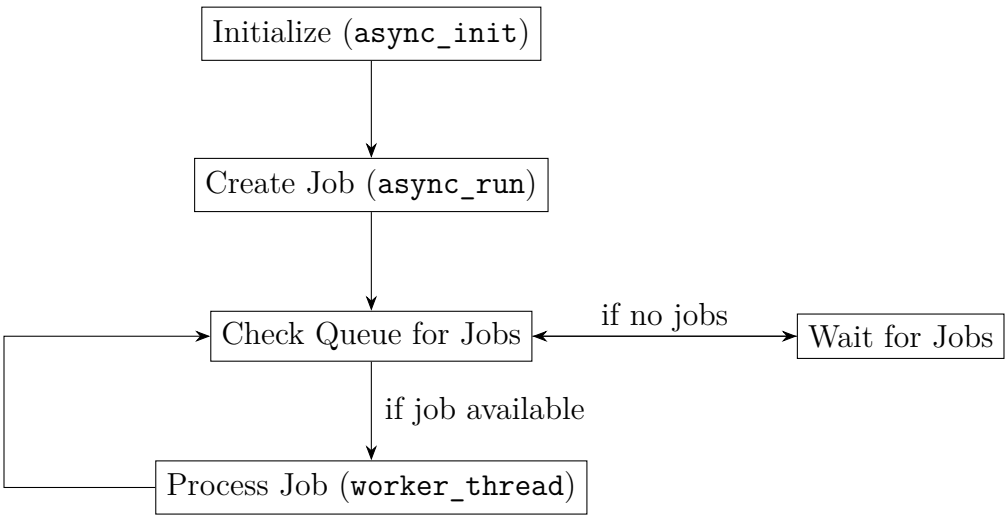


Figure 3: Program Framework of `async.c`

4.3 Design & Implementation

Execute jobs from the queue (`worker_thread` function)

The function is an infinite loop, consistently checking for jobs in the queue. Here is the principal segment of the `worker_thread`:

```
void *worker_thread(void *arg) {
    // ... (Initialization code)
    while (1) {
        pthread_mutex_lock(&job_queue->mutex);
```

```
        // ... (Code for job processing)
        pthread_mutex_unlock(&job_queue->mutex);
    }
    return NULL;
}
```

The thread locks the mutex to ensure that the operation is thread-safe. If there are no jobs in the queue, the thread will wait using a condition variable until a job becomes available. Once a job is available, it is removed from the queue, and the mutex is unlocked.

Initialize (`async_init` function)

It dynamically allocates memory for the job queue. It initializes the mutex and condition variable used in synchronization. And it creates a specified number of worker threads to process jobs. Each created thread starts executing the `worker_thread` function.

Submit jobs to the queue (`async_run` function)

This function accepts a handler function and its argument for job execution. It creates a job with the provided handler and arguments. It then locks the mutex to ensure that adding a job is a thread-safe operation. The job is added to the queue, and the condition variable is signaled to wake up any waiting threads. After that, the mutex is unlocked.

4.4 Program Result

HOW TO COMPILE: In the `thread_poll` directory, type `make` and press Enter in the console.
HOW TO EXECUTE: In the `thread_poll` directory, type `./httpserver --proxy inst.eecs.berkeley.edu:80 --port 8000 --num-threads 5` and press Enter in the console.
The screenshot of program result is shown in Figure 4 and Figure 5.

5 What did I learn

- Basic multi-threading knowledge
- Thread control and synchronization
- OS program framework design

```
Initializing with 5 threads...
Thread 1 has been created.
Thread 2 has been created.
Thread 3 has been created.
Thread 4 has been created.
Thread 5 has been created.
All threads initialized.
Listening on port 8000...

Starting Thread Number: 1
-----THREAD ID 281472793965024----- Initializing...
-----THREAD ID 281472793965024----- No jobs in the queue, waiting...

Starting Thread Number: 2
-----THREAD ID 281472802370016----- Initializing...
-----THREAD ID 281472802370016----- No jobs in the queue, waiting...

Starting Thread Number: 3
-----THREAD ID 281472785560032----- Initializing...
-----THREAD ID 281472785560032----- No jobs in the queue, waiting...

Starting Thread Number: 4
-----THREAD ID 281472819180000----- Initializing...
-----THREAD ID 281472819180000----- No jobs in the queue, waiting...

Starting Thread Number: 5
-----THREAD ID 281472810775008----- Initializing...
-----THREAD ID 281472810775008----- No jobs in the queue, waiting...
```

Figure 4: Initialization of the System

```
Accepted connection from 127.0.0.1 on port 53468
Added a new job. Total jobs in queue: 1
-----THREAD ID 281472793965024----- Processing a job...
Thread 281472793965024 will handle proxy request 0.
request thread 0 start to work
response thread 0 start to work
response thread 0 read failed, status 0
response thread 0 write failed, status 0
response thread 0 exited
Socket closed, proxy request 0 finished.

-----THREAD ID 281472793965024----- Completed the job.
-----THREAD ID 281472793965024----- No jobs in the queue, waiting...
Accepted connection from 127.0.0.1 on port 57564
Added a new job. Total jobs in queue: 1
-----THREAD ID 281472802370016----- Processing a job...
Thread 281472802370016 will handle proxy request 1.
request thread 1 start to work
response thread 1 start to work
response thread 1 read failed, status 0
response thread 1 write failed, status 0
response thread 1 exited
Socket closed, proxy request 1 finished.

-----THREAD ID 281472802370016----- Completed the job.
-----THREAD ID 281472802370016----- No jobs in the queue, waiting...
```

Figure 5: Output of the Worker Threads