Assignment2 Report

121090017

Zeng Guangjun

Frog crosses river

- **How did you design your program?**

Overview:

The game is set on a 2D grid (or map) where a frog tries to cross a river by hopping on logs that move horizontally. The frog starts at the bottom of the map and tries to reach the top without falling into the water or going beyond the boundary.

Detailed Design:

1.Structs & Global Variables:

struct Node: Represents the position of the frog on the grid with x and y coordinates.

The map array: Represents the game's grid. Rows are parts of the river or the banks, and columns are the width of the river. It holds characters to display either the river, logs, or the frog.

Global flags and variables (Out_detector and game_status) to help control and monitor the game's state.

2.Utility Functions:

kbhit(): Determines if a keyboard key has been pressed without blocking the program. It allows the game to continuously run while still checking for user inputs.

3.Game Display:

MapPrinter(): This function displays the game's map on the screen. The position of the logs is dynamic and changes over time, so the map is refreshed at regular intervals to reflect this.

4.Game Logic:

frog_move(): This is a thread function that handles the movement of the frog based on user input. It checks keyboard hits, updates the frog's position on the map, and checks the game's status (win, lose, or continue).

logs_move(): Another thread function that handles the movement of logs. It uses random starting positions for the logs and then continuously updates their position to simulate their movement in the river. It also checks if the frog is on a log and moves the frog accordingly.

5.Main Function:

Initializes the river map and the frog's starting position.

Launches two threads: one for moving the logs and another for controlling the frog.

Waits for both threads to finish, which occurs when the game ends.

Finally, based on the game's status (game_status), it displays the game's result: win, lose, or quit.

Program Flow:

1.Initialize the game's map and frog's starting position.

2.Start two threads:

One to handle the logs' movement (logs_move).

The other to handle the frog's movement (frog_move).

3.Both threads run concurrently. The logs move automatically, while the frog moves based on user input.

4.The game continues until the frog reaches the top (win), falls into the water (lose), or the user

decides to quit.
5.Once the game ends, the result is displayed to the user.

Note:
1.The game employs mutex locks (pthread_mutex_t lock) to ensure that shared resources, like the game's map, are accessed safely and without race conditions when updated by the two threads.
2.The game's display is frequently cleared and redrawn using escape sequences to give the appearance of real-time movement on the terminal.
3.The game uses some platform-specific functions (like tcgetattr, tcsetattr, and fcntl), which means it's primarily designed for Unix-like operating systems.

- **The environment of running your program. (E.g., version of OS and kernel)**

1. Version of OS

```
ubuntu@ubuntu-virtual-machine:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 16.04.7 LTS
Release:        16.04
Codename:       xenial
```

```
ubuntu@ubuntu-virtual-machine:~$ cat /etc/issue
Ubuntu 16.04.7 LTS \n \l
```

2. Version of kernel

```
ubuntu@ubuntu-virtual-machine:~$  uname -r
5.10.60
```

3. Version of gcc

```
ubuntu@ubuntu-virtual-machine:~$ 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.
```

- **The steps to execute your program**

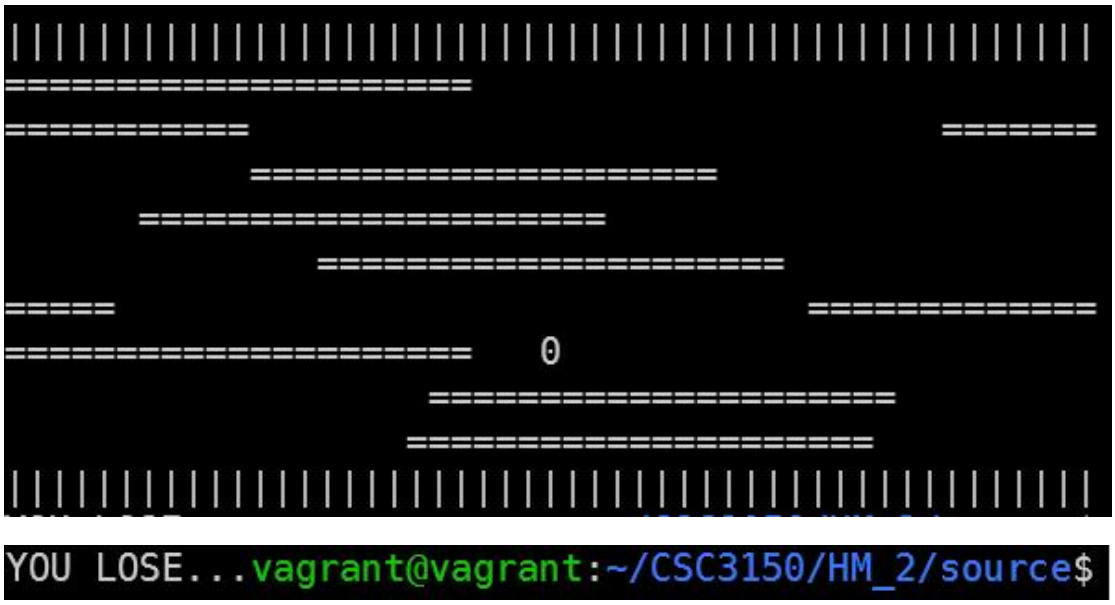
1. Copy the cpp file to the specified folder of the virtual machine
2. cd to this directory in the virtual machine
3. In the 'source' directory, type 'g++ hw2.cpp -lpthread' and enter on concole.
4. In the 'source' directory, type './a.out'.
5. Use key of ‘w,a,s,d’ to control the frog and use the key of ‘q’ to exit the game.

- **Screenshot of your program output.**

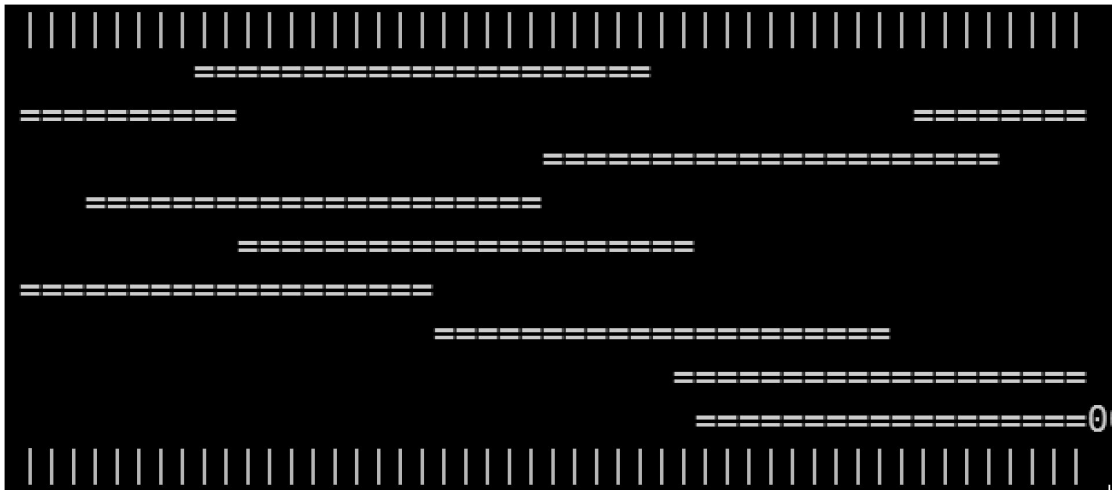
(1)win game:

(2) lose game:

(2a)Lose game because going into river



(2b)lose game because reaching the side of river (The range of movement of forg in the abscissa direction is 1-49, and when frog is not in bank and the abscissa is 0 or 50, it is determined to be out of bounds and lose game)



(3) quit game:

When the game runs normally, click the ‘q’ keyboard and quit the game.

- **What did you learn from the tasks?**

Multithreaded Programming: I learned the importance of handling multiple threads in a program. By implementing the movement of the logs and the frog, I deepened my understanding of how to maintain synchronization and avoid race conditions in real-time applications.

State Management: I understood the importance of maintaining and checking the game state. Deciding whether the player has won, lost, or quit the game helped me appreciate the need for robust state management in software development.

Concurrency with Mutex Locks: I delved into the world of mutex locks to ensure thread safety, especially when updating the positions of the logs and the frog. This taught me the practical applications of mutex locks in maintaining data integrity in multithreaded environments.

Edge Case Handling: Implementing rules like the frog's landing or the log reaching the river's side emphasized the importance of accounting for all possible scenarios in a program. This reinforced the idea that thorough testing and edge case consideration are crucial for any software solution.

Bonus Task

- **How did you design your program?**

Objective: The program appears to be a simple thread pool implementation that supports running tasks asynchronously using multiple threads. The idea is to allow a main program to submit tasks that get executed in the background, parallelizing work without the overhead of constantly creating and destroying threads.

Design Components:

1.Data Structures:

my_queue_t: Represents the task queue. It contains:

A doubly linked list of tasks.

A mutex for synchronization.

A condition variable to signal when a task has been added.

A size to keep track of the number of tasks.

my_item_t: Represents an individual task. It contains:

A function pointer called callback_task for the task to execute.

An integer argument to pass to the function.

Pointers for next and previous tasks, allowing it to be stored in a doubly linked list.

2.Thread Function (run_task):

This is the function executed by each worker thread.

It continuously checks if there are tasks in the queue. If the queue is empty, the thread waits on the condition variable.

Once a task is available, it pops the task from the front of the queue, then executes the task's callback with the provided argument.

3.Initialization (async_init):

Allocates and initializes the task queue.

Creates a specified number of worker threads (num_threads) which start running the run_task function.

4.Task Submission (async_run):

This function is used to submit a new task to the thread pool.

A new my_item_t (task) is created and initialized with the given function and argument.

The task is appended to the end of the task queue.

The condition variable is signaled to wake up any waiting threads.

Synchronization: Synchronization is crucial since multiple threads can access shared data (the task

queue). The following measures are implemented:

Mutex (my_thread_queue->mutex): Protects the task queue. Any operation that modifies or reads the task queue (e.g., adding/removing tasks, checking the size) is protected by locking this mutex.

Condition Variable (my_thread_queue->cond): Used to make worker threads wait when there are no tasks. When a new task is added, one waiting thread is signaled to wake up and process the task.

- **The environment of running your program. (E.g., version of OS and kernel)**

4. Version of OS

```
ubuntu@ubuntu-virtual-machine:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 16.04.7 LTS
Release:        16.04
Codename:       xenial
```

```
ubuntu@ubuntu-virtual-machine:~$ cat /etc/issue
Ubuntu 16.04.7 LTS \n \l
```

5. Version of kernel

```
ubuntu@ubuntu-virtual-machine:~$  uname -r
5.10.60
```

6. Version of gcc

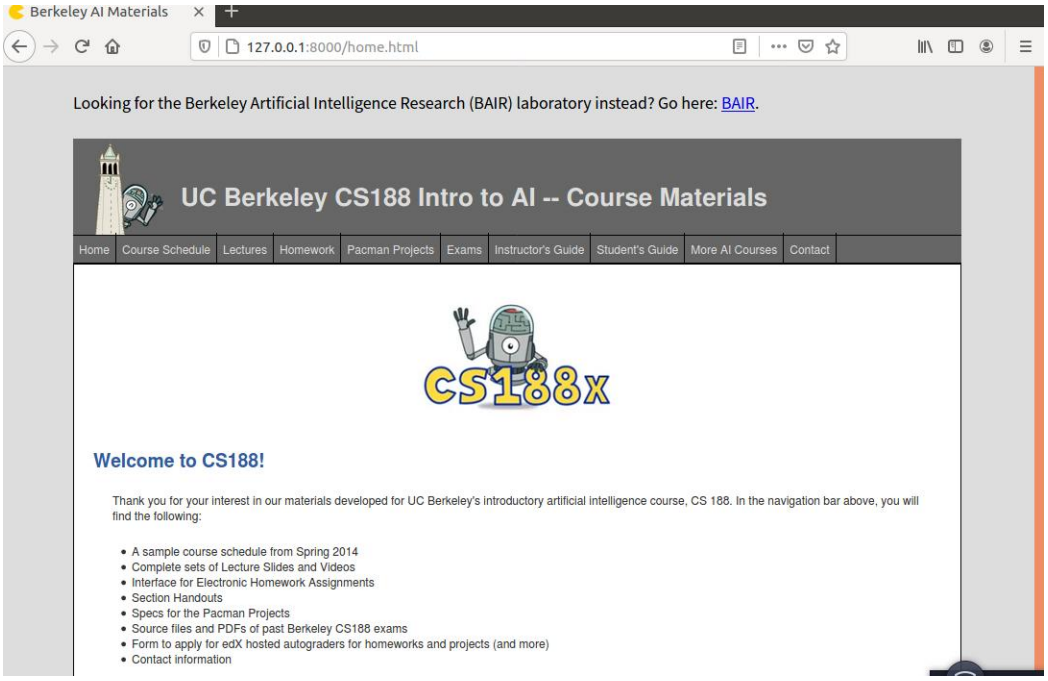
```
ubuntu@ubuntu-virtual-machine:~$ gcc --version
gcc (Ubuntu 5.4.0-6ubuntu1~16.04.12) 5.4.0 20160609
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

- **The steps to execute your program**

1. Copy the all source file to the specified folder of the virtual machine
2. cd to this directory in the virtual machine
3. In the 'source' directory, type ‘make’ in the terminal
4. In the 'source' directory, type ‘./httpserver --proxy inst.eecs.berkeley.edu:80 --port 8000 --num-threads 5’ in terminal
5. Enter ‘127.0.0.1:8000’ at my browser in the virtual machine
6. Click within the opened webpage to access the subpage

- **Screenshot of your program output.**

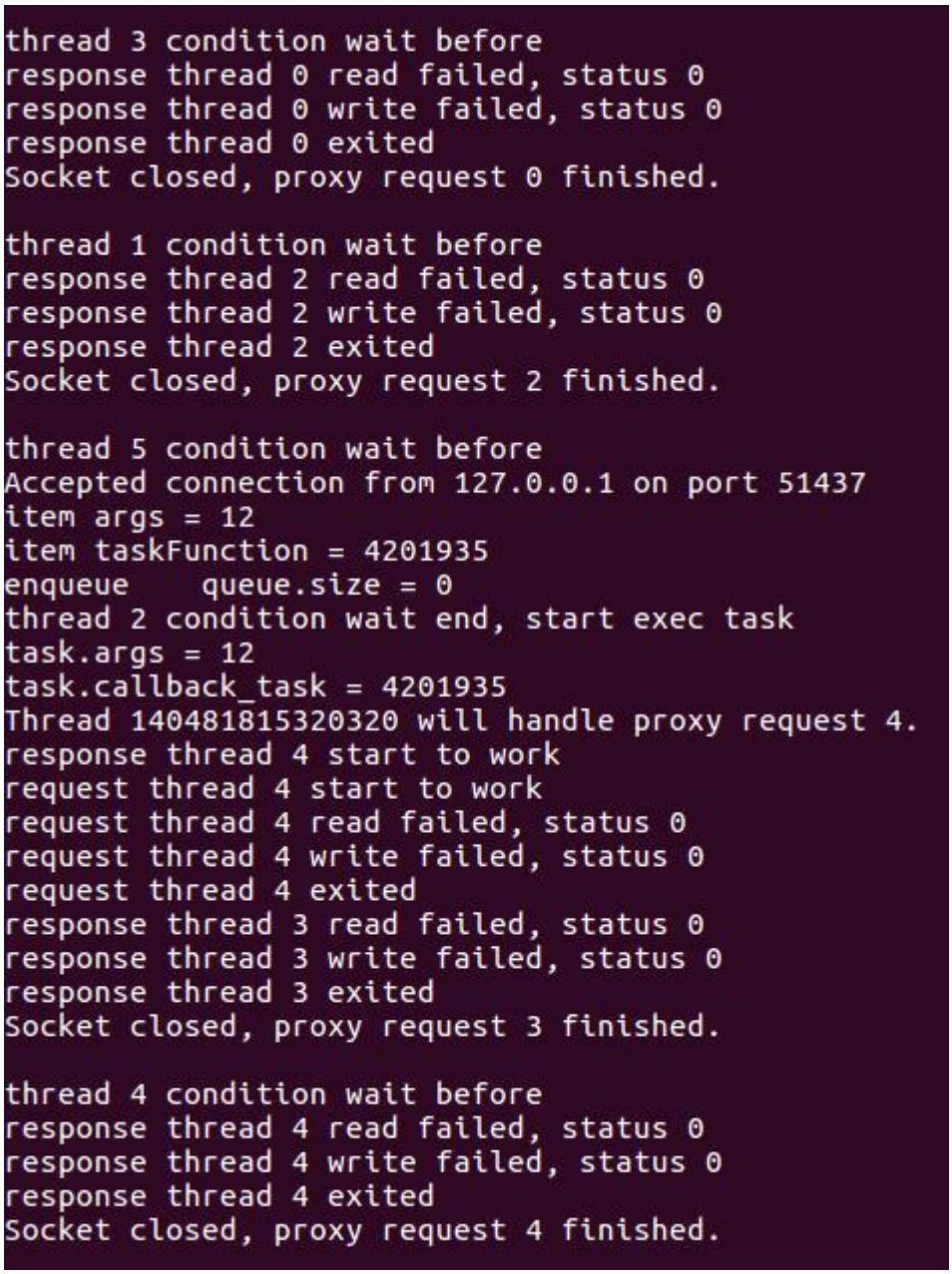
(1)the website



(2)the output in terminal

```
ubuntu@ubuntu-virtual-machine:~/bonus$ ./httpserver --proxy inst.eecs.berkeley.e
du:80 --port 8000 --num-threads 5
start 1
thread 1 condition wait before
start 3
thread 3 condition wait before
start 4
thread 4 condition wait before
start 5
thread 5 condition wait before
Listening on port 8000...
start 2
thread 2 condition wait before
Accepted connection from 127.0.0.1 on port 44269
item args = 4
item taskFunction = 4201935
enqueue    queue.size = 0
thread 1 condition wait end, start exec task
task.args = 4
task.callback_task = 4201935
Accepted connection from 127.0.0.1 on port 44781
item args = 5
item taskFunction = 4201935
enqueue    queue.size = 0
thread 3 condition wait end, start exec task
task.args = 5
task.callback_task = 4201935
Accepted connection from 127.0.0.1 on port 45293
item args = 8
item taskFunction = 4201935
enqueue    queue.size = 0
thread 4 condition wait end, start exec task
task.args = 8
task.callback_task = 4201935
Accepted connection from 127.0.0.1 on port 46317
item args = 10
item taskFunction = 4201935
enqueue    queue.size = 0
thread 5 condition wait end, start exec task
task.args = 10
task.callback_task = 4201935
```

```
enqueue    queue.size = 0
thread 5 condition wait end, start exec task
task.args = 10
task.callback_task = 4201935
Thread 140481823713024 will handle proxy request 0.
request thread 0 start to work
response thread 0 start to work
request thread 0 read failed, status 0
request thread 0 write failed, status 0
request thread 0 exited
Thread 140481806927616 will handle proxy request 1.
request thread 1 start to work
response thread 1 start to work
request thread 1 read failed, status 0
request thread 1 write failed, status 0
request thread 1 exited
Thread 140481790142208 will handle proxy request 2.
request thread 2 start to work
request thread 2 read failed, status 0
request thread 2 write failed, status 0
request thread 2 exited
response thread 2 start to work
Thread 140481798534912 will handle proxy request 3.
response thread 3 start to work
request thread 3 start to work
request thread 3 read failed, status 0
request thread 3 write failed, status 0
request thread 3 exited
response thread 1 write failed, status -1
response thread 1 exited
Socket closed, proxy request 1 finished.
```



- **What did you learn from the tasks?**

Efficient Thread Management: I will understand the importance of reusing threads instead of constantly creating and destroying them. This will teach me how to design applications that are both resource-efficient and high-performing.

Synchronization Skills: I'll gain hands-on experience with thread synchronization, using mutexes and condition variables. This will bolster my ability to write bug-free, concurrent code that works harmoniously without race conditions.

Real-world Application Insights: By integrating my thread pool with an HTTP server, I'll get a tangible sense of the real-world implications of my coding decisions. This will be a rewarding experience that ties theory to practice.

Enhanced Debugging Techniques: I'll be challenged to debug multithreaded code, which can be notoriously tricky. This will sharpen my debugging skills and deepen my appreciation for the

intricacies of concurrent programming.