CSC 3150 Project #2

Ziyu XIE
121090642

October 19, 2023

Contents

1 Introduction

1.1 Project tasks

This project requires us to create a frog-cross-river game. A river has logs floating on it, and a frog must cross the river by jumping on the logs as they pass by via clicking ”W”, ”S”, ”A”, ”D”. This game should output ”win”, ”loss”, or ”quit” status when different situations happen. To make such game in terminal using c++, We need to set the correct Linux environment and some knowledge about the multi-thread processing.

Further, the bonus task asks us to implement a thread pool, which can handle the request concurrently.

In this project, I have acheived both of them.

1.2 Development environment

The development environment is Ubuntu 16.04 with Linux kernel version 5.10.197. The gcc version is 5.4.0.

To set up this environment, we should first get the correct Linux kernel version installed. Follow the command lines below to get the kernel version 5.10.197.

2 Implementing of the frog-cross-river game

2.1 Basic design

To finish this project, we need some units like the frog and the maps. We use a 2D array to store the map, and we use a structure to store the frog. Then, we should create pthreads for logs, and we should also create some pthreads to control the screen and the game.

For the logs threads, we should let the logs to move automatically and detect whether there is a keyboard clicking. If clicking, we should process the frog moving.

For the control threads, we should refresh the screen with a high frequency and judge the game status.

The details will be shown later.

2.2 Global preparations

In this part, I firstly define the constant numbers, including the row, column, length of logs, and the delay time of log and print.

Then I define the struct for the frog, the char-2D- array-based map, the register to record the center of each log, and the game status. In addtion, there is a mutex to protect the map.

The details are shown below.

```
/* const numbers define */
#define ROW 10
#define COLUMN 50
#define LOG_LEN 15
#define LOG_DELAY 120000
#define PRINT_DELAY 13000

/* struct for the frog */
struct Node
{
    int x, y;
    Node(int _x, int _y) : x(_x), y(_y){};
    Node(){};
} frog;

/* global variables */
char map[ROW + 10][COLUMN];
int log_center[ROW + 1];
int game_status; // 0 for run, 1 for win, 2 for lose, 3 for quit

/* global pthread variables */
pthread_mutex_t wood_lock;

/* useful helping functions */
int kbhit(void);
void map_print(void);
int rand_num_gen(void);
int get_mod(int, int);
/* logs helping functions */
void logs_shifting(int, int);
void logs_initial(int);
/* frog move helping functions */
void frog_move_help(char ch);
/* pthread functions */
void *screen_print(void *t);
void *logs_move(void *t);
```

2.3 Implementing the main function

The main functions can be divided into three parts, the initialization, the pthreads creation, and the pthreads join.

The firt part is to initialize the map and status. We should use the `printf("\033[2J")` to clean the terminal. And we should also initialize the mutex.

```
/* Initialize the river map and frog's starting position */
memset(map, 0, sizeof(map)); // set all the chars(1 byte each)
    of the map to '0'
for (i = 1; i < ROW; ++i) // create the empty river
{
    for (j = 0; j < COLUMN - 1; ++j)
        map[i][j] = ' ';
}
for (j = 0; j < COLUMN - 1; ++j) // create the bottom bank with
    len (COLUMM - 1)
    map[ROW][j] = map[0][j] = '|';
for (j = 0; j < COLUMN - 1; ++j) // create the top bank with
    len (COLUMM - 1)
    map[0][j] = map[0][j] = '|';
frog = Node(ROW, (COLUMN - 1) / 2); // create the frog
map[frog.x][frog.y] = '0';     // put the frog on the map

/* clean the terminal and set game status */
printf("\033[2J");
map_print();
game_status = 0;

/* Initialize the mutex for wood move */
pthread_mutex_init(&wood_lock, NULL);
```

The second part is to create the pthreads. We should create the pthreads for logs, the pthreads for control, and the pthreads for screen.

```
/* Create pthreads for wood move and frog control. */
pthread_t wood_threads[ROW]; // 0 for frog, others for wood
for (i = 1; i <= ROW - 1; i++)
    pthread_create(&wood_threads[i], NULL, logs_move, (void
        *)(long)i);

/* Create pthreads for screen print and game status */
pthread_t control_threads[2]; // 0 for print, and 1 for status
pthread_create(&control_threads[0], NULL, screen_print, NULL);
pthread_create(&control_threads[1], NULL, status_judge, NULL);
```

The third part is to join the pthreads and output the result based on the game status. Whenever the game status changes to nonzero, all the pthreads created will exit.

```c
/* join all the threads */
for (i = 1; i <= ROW - 1; i++)
    pthread_join(wood_threads[i], NULL);
for (i = 0; i <= 1; i++)
    pthread_join(control_threads[i], NULL);
/* Display the output for user: win, lose or quit. */
printf("\033[2J"); // clean the terminal
printf("\033[H"); // move the cursor to the top left corner
switch (game_status)
{
case 1:
    printf("You win the game!!\n");
    break;
case 2:
    printf("You lose the game!!\n");
    break;
case 3:
    printf("You exit the game!!\n");
    break;
default:
    break;
}
/* exit the pthread */
pthread_exit(NULL);
return 0;
```

2.4 Implementing useful helping functions

The four useful helping functions are `int kbhit(void)`, `void map_print(void)`, `rand_num_gen(void)`, and `get_mod(int, int)`.

The `int kbhit(void)` function is used to detect whether there is a keyboard clicking. If yes, it will return 1, otherwise it will return 0.

The `void map_print(void)` function is used to print the map. It will firstly lock the mutex, and then print the map. After printing, it will unlock the mutex.

The `rand_num_gen(void)` function is used to generate a random number. It will use the `time()` function to get the current time, and then use the `srand()` function to set the seed. Finally, it will use the `rand()` function to generate a random number.

The `get_mod(int, int)` function is used to get the mod of two integers. It will firstly get the mod, and then judge whether it is negative. If yes, it will add the divisor to it. This function will be used frequently in the following codes.

The details of these codes are attached and are not be shown in the report.

2.5 Implementing the logs pthreads

The logs pthreads are used to control the logs. It will firstly get the current time, and then use the `rand_num_gen()` function to generate a random number. This random number will be used to initialize the places of each log. The logs will move to the right, or to the left based on the row value. Then it uses the function `void logs_shifting(int, int)` to move the logs.

The details are shown below.

```c
/* shift the logs in map */
void logs_shifting(int row, int direc)
{
    int left_end;
    int right_end;
    int center = get_mod(log_center[row], COLUMN - 1);
    int half_log = LOG_LEN / 2;

    switch (direc)
    {
    case 1: // left
        left_end = get_mod(center - half_log - 1, COLUMN - 1);
        right_end = get_mod(center + half_log, COLUMN - 1);
        map[row][left_end] = '=';
        map[row][right_end] = ' ';
        if (is_frog_on_log(row, frog.x, frog.y))
        {
            map[row][frog.y] = '=';
            if (frog.y == right_end)
                map[row][frog.y] = ' ';
            frog.y = get_mod(frog.y - 1, COLUMN - 1);
            map[row][frog.y] = '0';
        }
        log_center[row] = get_mod(center - 1, COLUMN - 1);
        break;
    case 0: // right
        left_end = get_mod(center - half_log, COLUMN - 1);
        right_end = get_mod(center + half_log + 1, COLUMN - 1);
        map[row][left_end] = ' ';
        map[row][right_end] = '=';
        if (is_frog_on_log(row, frog.x, frog.y))
        {
            map[row][frog.y] = '=';
            if (frog.y == left_end)
                map[row][frog.y] = ' ';
            frog.y = get_mod(frog.y + 1, COLUMN - 1);
            map[row][frog.y] = '0';
        }
        log_center[row] = get_mod(center + 1, COLUMN - 1);
        break;
    default:
        break;
    }
}
```

In the logs pthreads, if we click the keyboard, it will go to the process of moving frog. We should use kbhit to detect it and use the function `void frog_move_help(char ch)` to update the place of the frog.

In details, we should judge the index of the frog for different updating. The details are shown below.

```c
/* help moving the frog */
void frog_move_help(char ch)
{
    int old_x = frog.x;
    int old_y = frog.y;
    int half_log = LOG_LEN / 2;
    int center = log_center[old_x];
    /* process previous entry */
    if (old_x == ROW || old_x == 0) // frog is on the bottom
        bank
        map[old_x][old_y] = '|';
    else if (old_x != ROW && old_x != 0) // frog is on the river
    {
        if (is_frog_on_log(old_x, old_x, old_y))
            map[old_x][old_y] = '=';
        else
            map[old_x][old_y] = ' ';
    }
    /* move the frog index */
    switch (ch)
    {
    case 'W': case 'w':
        if (frog.x != 0)
            frog.x -= 1;
        break;
    case 'S': case 's':
        if (frog.x != ROW)
            frog.x += 1;
        break;
    case 'A': case 'a':
        if (frog.y != 0)
            frog.y -= 1;
        break;
    case 'D': case 'd':
        if (frog.y != COLUMN - 2)
            frog.y += 1;
        break;
    case 'Q': case 'q':
        game_status = 3;
        break;
    default:
        break;
    }
    /* set new frog on map */
    map[frog.x][frog.y] = '0';
}
```

2.6 Implementing the control threads

The control threads include two parts, one for printing and another for game status judging.

For the printing, we should frequently refresh the terminal, i.e. print the latest map. This thread will exit when the game status changes.

For the game status, whenever the game reaches the condition to finish, it will change the game status and exit.

The details for the game status thread are shown below.

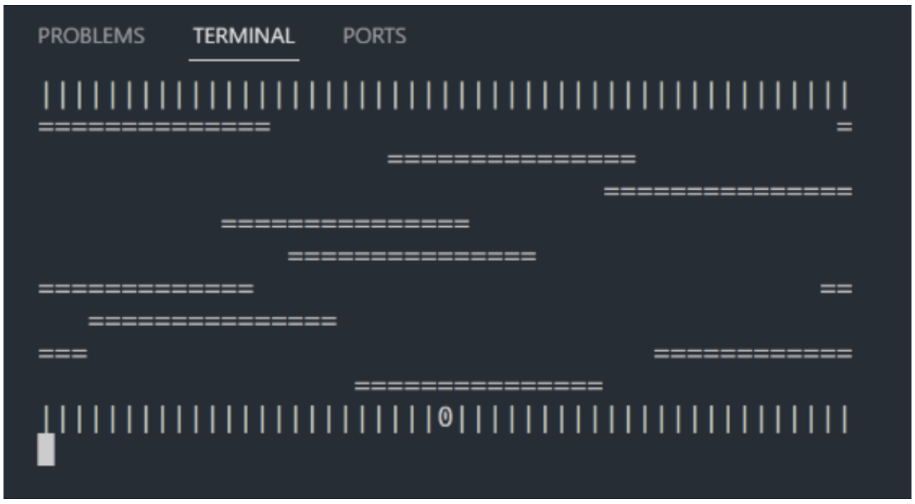
```
/* judge the status all the time */
void *status_judge(void *t)
{
    while (1)
    {
        pthread_mutex_lock(&wood_lock);
        /* Check game's status */
        if (game_status != 0)
        {
            pthread_mutex_unlock(&wood_lock);
            pthread_exit(NULL);
        }
        /* Check if the frog is on the top bank */
        if (frog.x == 0)
        {
            game_status = 1; // win the game
            pthread_mutex_unlock(&wood_lock);
            pthread_exit(NULL);
        }
        /* Check if the frog touches the edge */
        else if (frog.y == 0 || frog.y == COLUMN - 2)
        {
            game_status = 2; // lose the game
            pthread_mutex_unlock(&wood_lock);
            pthread_exit(NULL);
        }
        /* Check if the frog is on the river */
        else if (frog.x != ROW && frog.x != 0)
        {
            if (!is_frog_on_log(frog.x, frog.x, frog.y))
            {
                game_status = 2; // lose the game
                pthread_mutex_unlock(&wood_lock);
                pthread_exit(NULL);
            }
        }
        pthread_mutex_unlock(&wood_lock);
    }
}
```

2.7 Executing and output

To execute the file, you can follow the command line below.

```
g++ hw2.cpp -lpthread
./a.out
```

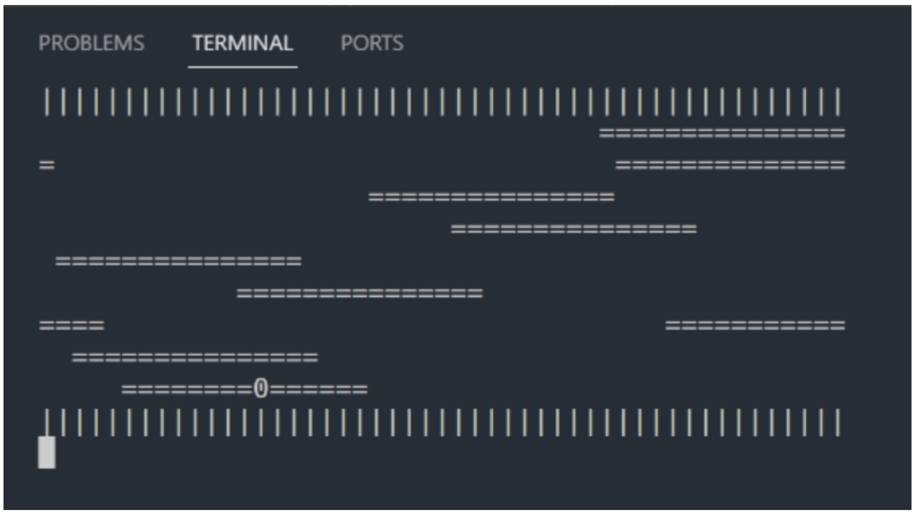
And then you will see the screen like this:



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

Figure 1: Starting screen

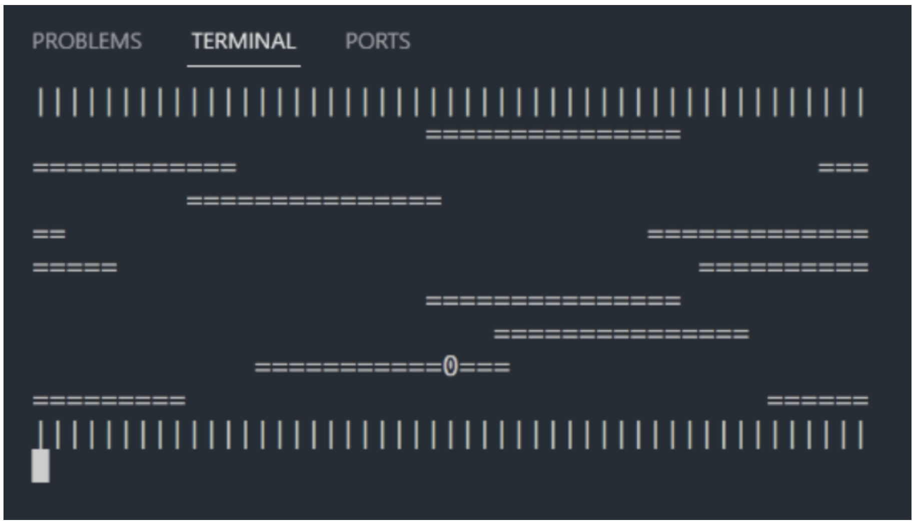
Then you can go up, down, left, right when you type ”W”, ”S”, ”A”, ”D”. You need to reach each logs step by step, shown in Figure 2, 3.



PROBLEMS TERMINAL PORTS

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

Figure 2: Moving the frog - 1

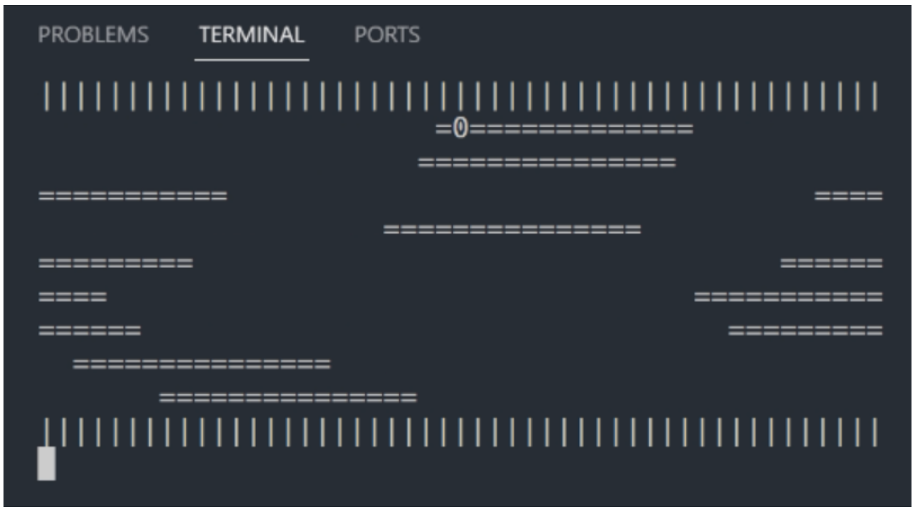


PROBLEMS TERMINAL PORTS

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

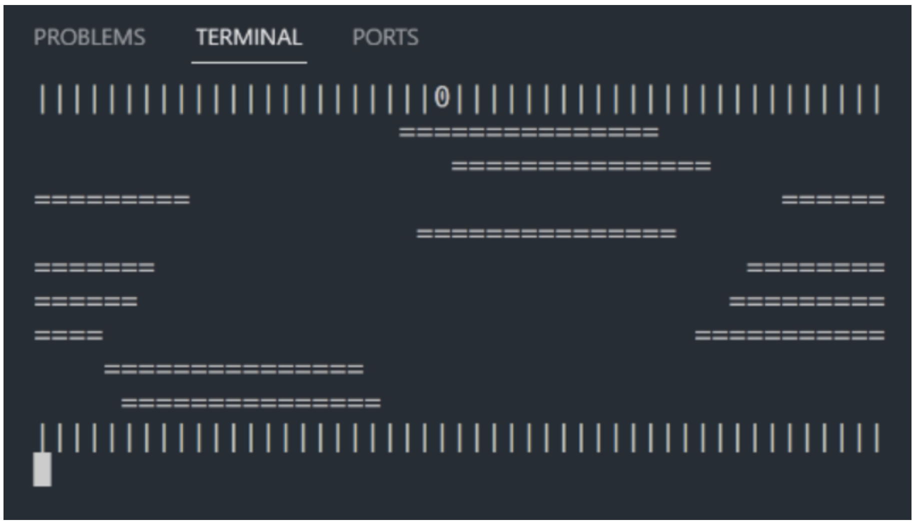
Figure 3: Moving the frog - 2

Continue to do so, If you successfully reach the top bank, you will win the game, shown in Figure 4, 5, 6.



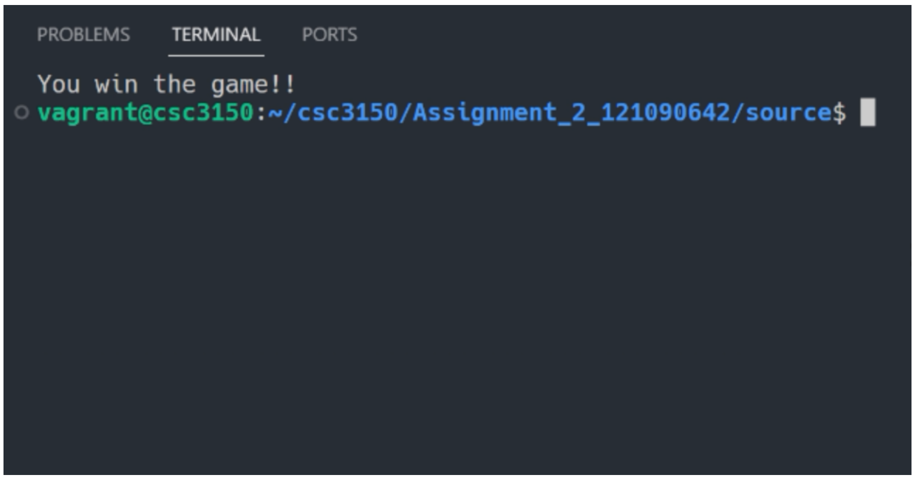
```
|||||||||||||||||||||||||||||||||||||||||||||||||
                       =0=============
                      ===============
===========                                  ====
                    ===============
=========                                  ======
====                             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

Figure 4: Win the game - 1



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

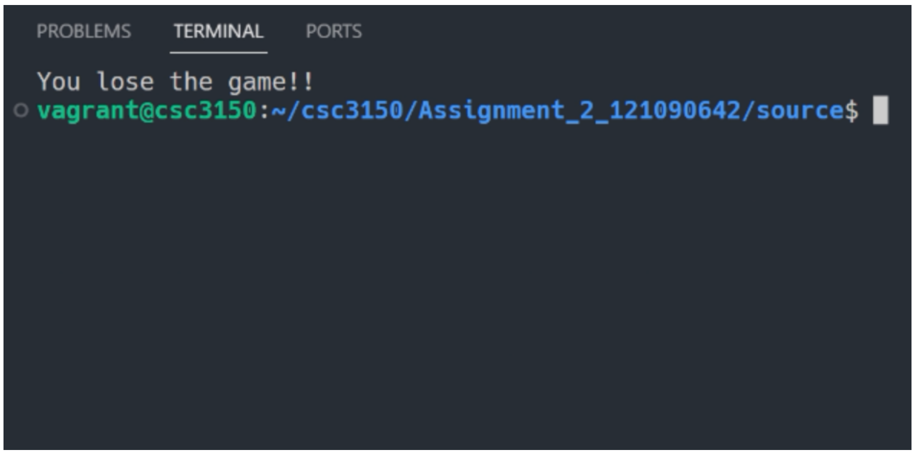
Figure 5: Win the game - 2



```
You win the game!!
vagrant@csc3150:~/csc3150/Assignment_2_121090642/source$
```

Figure 6: Win the game - 3

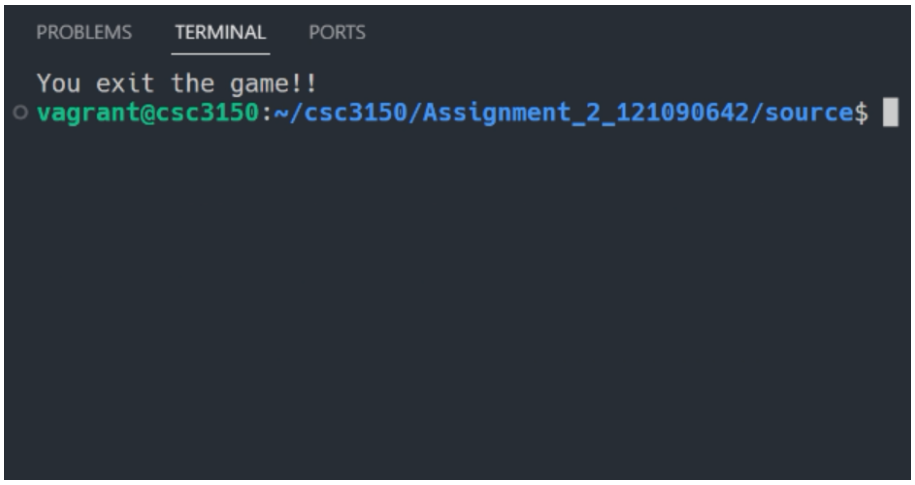
If you unfortunately touch the boundary or drop into the water, you will lose the game, as shown in Figure 7.



```
You lose the game!!
vagrant@csc3150:~/csc3150/Assignment_2_121090642/source$
```

Figure 7: Lose the game

If you type ”Q”, you will quit the game, as shown in Figure 8.



```
You exit the game!!
vagrant@csc3150:~/csc3150/Assignment_2_121090642/source$
```

Figure 8: Exit the game

Here just shown some screenshots of important points. For the further understanding of the game, you can execute the game and play it by yourself.

3 Implementing of the bonus task

3.1 Basic design

This program is asking us to implement a thread pool, which can handle the request concurrently. To handle these requests, we can use a linked queue to connect some pthreads together, so that each time the pthread id will be different.

In additon, we need the condition variables and the mutex to protect and control the pool.

The details will be shown later.

3.2 Implementing the queue and item

The struct my_item is used to describe the job given by the client. It includes the function pointer and the argument. In addition, in order to connect the items together, we should add a pointer to the next item and the previous one.

The struct my_queue is just designed as a pool. It can store the head and the tail of the pool, so that it can keep all the information of the tasks. This pool should keep the pthreads in order to execuate the tasks. In addition, it should store the current number in order to wait if there is no task. Furthermore, the two condition variables and one mutex is used to wait the pthreads conditionally.

The details are shown below.

```c
typedef struct my_item
{
    void (*hanlder)(int);
    int args;
    struct my_item *next_pt;
    struct my_item *prev_pt;

} my_item_t;

typedef struct my_queue
{
    pthread_t *pthreads; // pthreads
    int cur_num;      // current number of queue

    my_item_t *head; // head of queue
    my_item_t *tail; // tail of queue

    pthread_mutex_t mutex;  // mutex
    pthread_cond_t empty_true; // queue is empty
    pthread_cond_t empty_false; // queue is not empty

} my_queue_t;
```

3.3 Implementing the init function

The init function is used to initialize the pool. Since the pool should control all the pthreads, in my code, the pool has been global declared and locally defined in the init function.

After defined the primal pool, we should also initialize the mutex and the condition variables. Then we create the pthreads.

Here we should NOT join them and destroy the mutex and the condition variables because the pthreads serve forever in this project. If we do so, this initialization won't be finished and hence cannot start doing the tasks.

```c
void async_init(int num_threads)
{
    pool = (my_queue_t *)malloc(sizeof(my_queue_t));
    pool->cur_num = 0;
    pool->head = NULL;
    pool->tail = NULL;

    // printf("[Init] : Inialializing mutex and conditon
        variables...\n");
    pthread_mutex_init(&pool->mutex, NULL);
    pthread_cond_init(&pool->empty_true, NULL);
    pthread_cond_init(&pool->empty_false, NULL);

    // printf("[Init] : Initializing threads...\n");
    pool->pthreads = (pthread_t *)malloc(sizeof(pthread_t) *
        num_threads);

    for (int i = 0; i < num_threads; i++)
        // printf("[Init] : Creating pthread [%d]...\n", i);
        pthread_create(&(pool->pthreads[i]), NULL, foo, (void
            *)i);
    // printf("[Init] : Initialize the pool successfully!\n");

    return;
}
```

3.4 Implementing the executing function

The foo function is served as the executing function of the task. It will firstly lock the mutex, and then get the task from the pool. If there is no task, it will wait for the condition variable. If there is a task, it will execute the task and unlock the mutex.

It will also update the pool chain so that this pool will not break down. If the pool is empty, it will broadcast the signal of empty_true. If the pool is not empty, it will signal the empty_false. These signals will be used in the next run function in order to keep the item tasks run orderly.

```c
void *foo(void *t)
{
    /* allocate memory for pointer */
    // printf("[foo%d] : Allocate memory for pointer\n", t);
    my_item_t *web_job = NULL;

    while (1)
    {
        pthread_mutex_lock(&pool->mutex);

        while (pool->cur_num == 0)
        {
            // printf("[foo%d] : No job in the pool, waiting for it.\n", t);
            pthread_cond_wait(&pool->empty_false, &pool->mutex);
        }

        // printf("[foo%d] : Fetching the job...\n", t);
        web_job = pool->head; // fetch the job
        // printf("[foo%d] : Fetch the job successfully!\n", t);

        /* after fetching, update the pool */
        // printf("[foo%d] : The current num of pool is %d.\n", t, pool->cur_num);
        if (web_job->next_pt == NULL) // if the queue will be empty
        {
            // printf("[foo%d] : The queue will be empty.\n", t);
            pool->head = NULL; // empty the head
            pool->tail = NULL; // empty the tail
            pthread_cond_broadcast(&pool->empty_true);
        }
        else // if the queue is not empty
        {
            // printf("[foo%d] : The queue is still not empty.\n", t);
            pool->head = web_job->next_pt; // update the head item
            web_job->next_pt->prev_pt = NULL; // update the head's prev
            pthread_cond_broadcast(&pool->empty_false);
        }
        // printf("[foo%d] : Update the pool successfully!\n", t);

        pool->cur_num--; // update the queue number

        pthread_mutex_unlock(&pool->mutex);
        web_job->hanlder(web_job->args); // do the job

        /* free the memory of the job */
        web_job = NULL;
        // printf("[foo%d] : Job %d has been done!\n", t);
    }
}
```

3.5 Implementing the run function

The run function is served to add the tasks to the pool in order to excute them.

We can initialize an item and store the information of the new task. Then we update the pool chain and signal the empty_false in order to wait the pthreads which are waiting for this signal. Finally, we unlock the mutex.

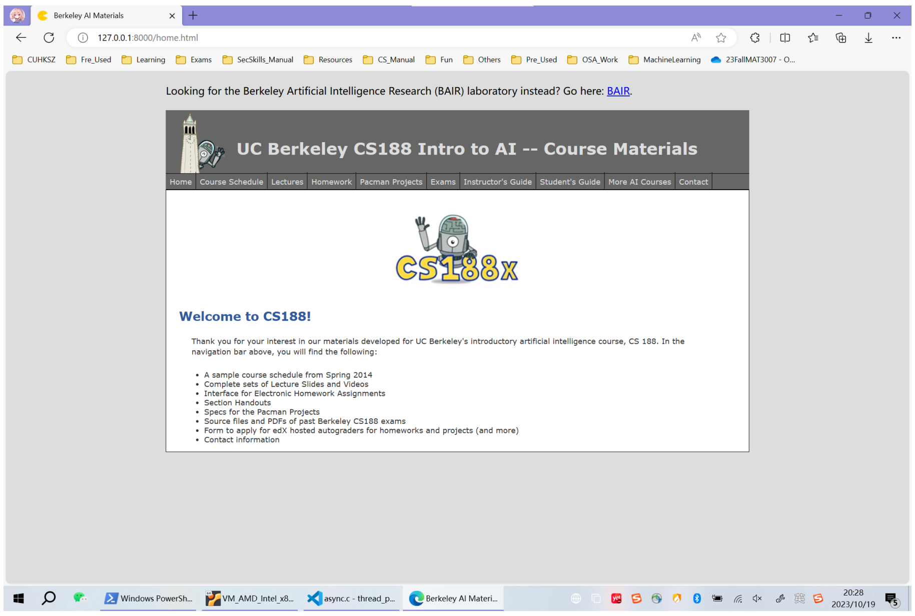
```
void async_run(void (*hanlder)(int), int args)
{
    pthread_mutex_lock(&pool->mutex);

    // printf("[Run] : Get a new job...\n");

    /* create a new job pointer, allocate memory */
    my_item_t *web_job = (my_item_t *)malloc(sizeof(my_item_t));
    web_job->hanlder = hanlder;
    web_job->args = args;

    // printf("[Run] : Update the pool...\n");
    // printf("[Run] : The current num of pool is %d.\n",
        pool->cur_num);
    /* update the pool */
    if (pool->head == NULL) // if the queue is empty
    {
        // printf("[Run] : No job in the pool.\n");
        pool->head = web_job;
        pool->tail = web_job;
        // printf("[Run] : Set head = tail = job.\n");
    }
    else // if the queue is not empty
    {
        // printf("[Run] : Some jobs in the pool.\n");
        web_job->prev_pt = pool->tail;
        pool->tail->next_pt = web_job;
        pool->tail = web_job;
        // printf("[Run] : Update the pool.\n");
    }

    pthread_cond_broadcast(&pool->empty_false);
    // printf("[Run] : Pool in the job successfully!\n");

    /* update the queue number */
    pool->cur_num++;
    // printf("[Run] : The current num of pool is %d.\n",
        pool->cur_num);

    pthread_mutex_unlock(&pool->mutex);

    return;
}
```

3.6 Executing and output

To execute the file, you can follow the command line below.

```
make
./httpserver --proxy inst.eecs.berkeley.edu:80 --port 8000
    --num-threads 5
```

Then we will see the word "Listening on port 8000...". After this, we access 127.0.0.1:8000 in the browser, and we will see the screen look like Figure 9:



127.0.0.1:8000/home.html

Looking for the Berkeley Artificial Intelligence Research (BAIR) laboratory instead? Go here: BAIR.

UC Berkeley CS188 Intro to AI -- Course Materials

Home | Course Schedule | Lectures | Homework | Pacman Projects | Exams | Instructor's Guide | Student's Guide | More AI Courses | Contact

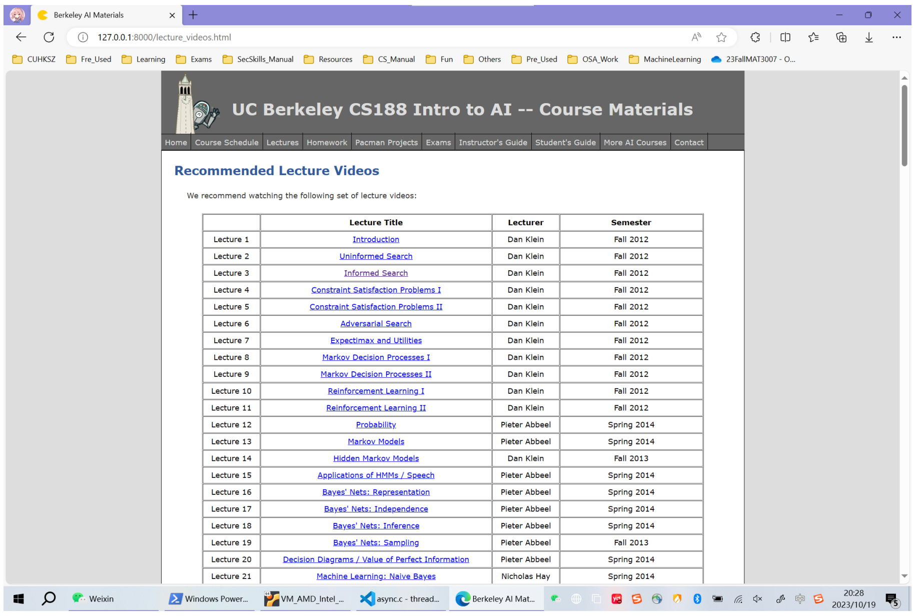
Welcome to CS188!

Thank you for your interest in our materials developed for UC Berkeley's introductory artificial intelligence course, CS 188. In the navigation bar above, you will find the following:

- A sample course schedule from Spring 2014
- Complete sets of Lecture Slides and Videos
- Interface for Electronic Homework Assignments
- Section Handouts
- Specs for the Pacman Projects
- Source files and PDFs of past Berkeley CS188 exams
- Form to apply for edX hosted autograders for homeworks and projects (and more)
- Contact information

Figure 9: Browser screen - 1

Randomly click some links in this website. Here I just shown a few of them which are shown in Figure 10 and Figure 11, you can click even more.



127.0.0.1:8000/lecture_videos.html

UC Berkeley CS188 Intro to AI -- Course Materials

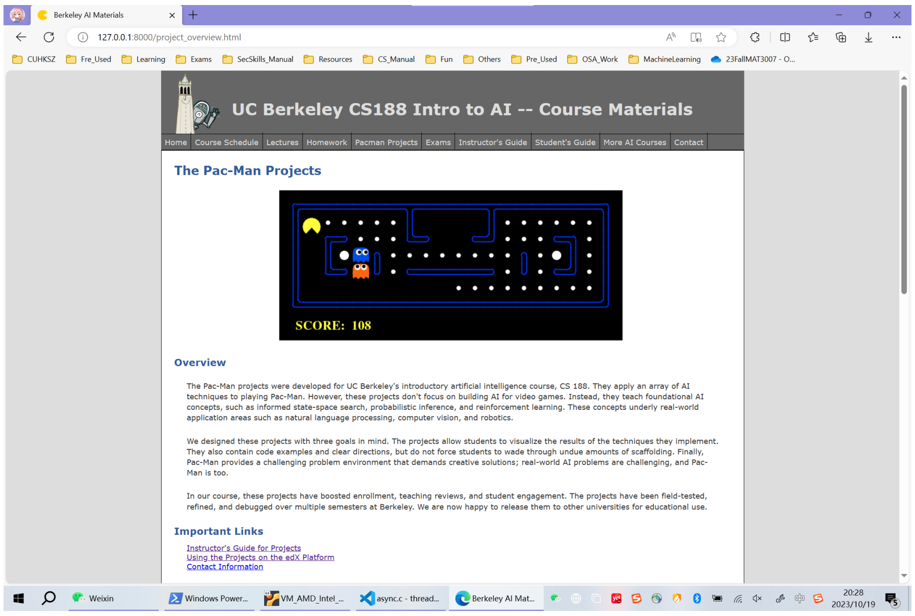
Home | Course Schedule | Lectures | Homework | Pacman Projects | Exams | Instructor's Guide | Student's Guide | More AI Courses | Contact

Recommended Lecture Videos

We recommend watching the following set of lecture videos:

	Lecture Title	Lecturer	Semester
Lecture 1	Introduction	Dan Klein	Fall 2012
Lecture 2	Uninformed Search	Dan Klein	Fall 2012
Lecture 3	Informed Search	Dan Klein	Fall 2012
Lecture 4	Constraint Satisfaction Problems I	Dan Klein	Fall 2012
Lecture 5	Constraint Satisfaction Problems II	Dan Klein	Fall 2012
Lecture 6	Adversarial Search	Dan Klein	Fall 2012
Lecture 7	Expectimax and Utilities	Dan Klein	Fall 2012
Lecture 8	Markov Decision Processes I	Dan Klein	Fall 2012
Lecture 9	Markov Decision Processes II	Dan Klein	Fall 2012
Lecture 10	Reinforcement Learning I	Dan Klein	Fall 2012
Lecture 11	Reinforcement Learning II	Dan Klein	Fall 2012
Lecture 12	Probability	Pieter Abbeel	Spring 2014
Lecture 13	Markov Models	Pieter Abbeel	Spring 2014
Lecture 14	Hidden Markov Models	Dan Klein	Fall 2013
Lecture 15	Applications of HMMs / Speech	Pieter Abbeel	Spring 2014
Lecture 16	Bayes' Nets: Representation	Pieter Abbeel	Spring 2014
Lecture 17	Bayes' Nets: Independence	Pieter Abbeel	Spring 2014
Lecture 18	Bayes' Nets: Inference	Pieter Abbeel	Spring 2014
Lecture 19	Bayes' Nets: Sampling	Pieter Abbeel	Fall 2013
Lecture 20	Decision Diagrams / Value of Perfect Information	Pieter Abbeel	Spring 2014
Lecture 21	Machine Learning: Naive Bayes	Nicholas Hay	Spring 2014

Figure 10: Browser screen - 2



127.0.0.1:8000/project_overview.html

UC Berkeley CS188 Intro to AI -- Course Materials

Home | Course Schedule | Lectures | Homework | Pacman Projects | Exams | Instructor's Guide | Student's Guide | More AI Courses | Contact

The Pac-Man Projects

SCORE: 108

Overview

The Pac-Man projects were developed for UC Berkeley's introductory artificial intelligence course, CS 188. They apply an array of AI techniques to playing Pac-Man. However, these projects don't focus on building AI for video games. Instead, they teach foundational AI concepts, such as informed state-space search, probabilistic inference, and reinforcement learning. These concepts underly real-world application areas such as natural language processing, computer vision, and robotics.

We designed these projects with three goals in mind. The projects allow students to visualize the results of the techniques they implement. They also contain code examples and clear directions, but do not force students to wade through undue amounts of scaffolding. Finally, Pac-Man provides a challenging problem environment that demands creative solutions; real-world AI problems are challenging, and Pac-Man is too.

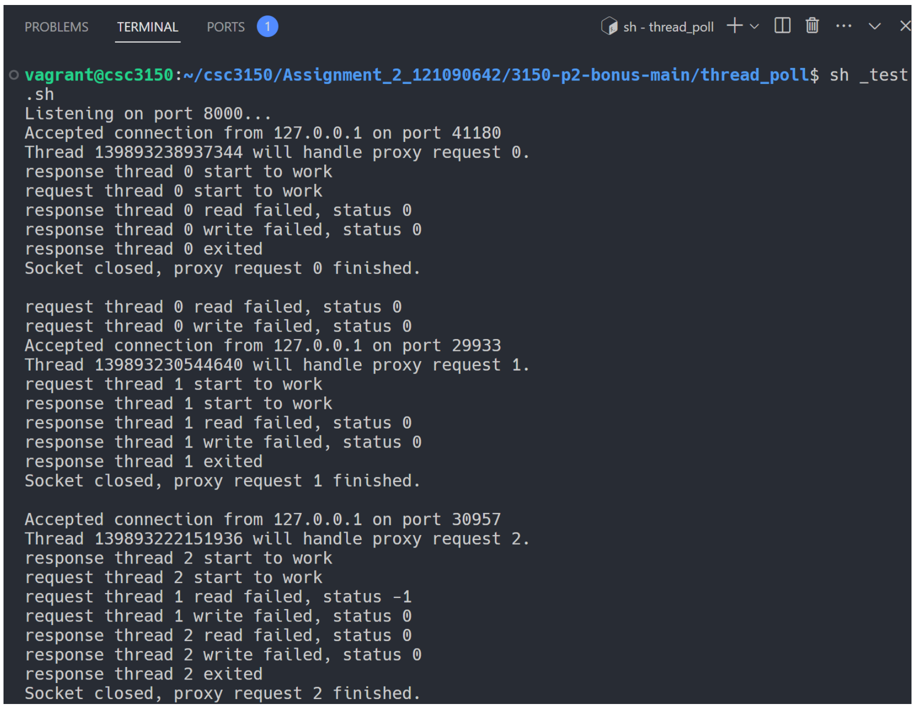
In our course, these projects have boosted enrollment, teaching reviews, and student engagement. The projects have been field-tested, refined, and debugged over multiple semesters at Berkeley. We are now happy to release them to other universities for educational use.

Important Links

Instructor's Guide for Projects
Using the Projects on the edX Platform
Contact Information

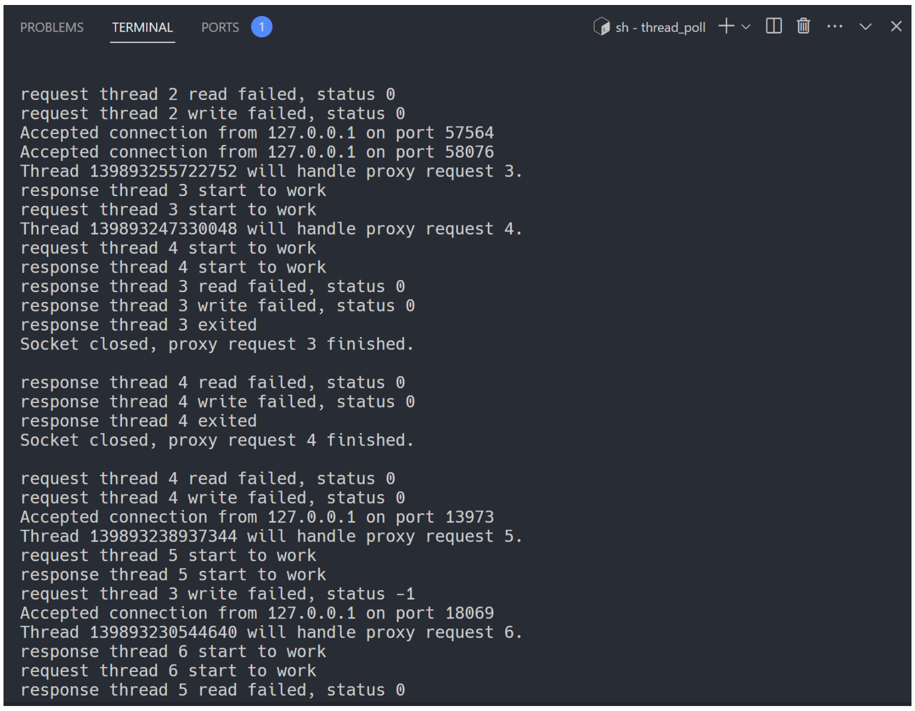
Figure 11: Browser screen - 3

After that, go back to the terminal and we can see these things which are shown in Figure 12 - 14, which shows that the requests have been handled concurrently.



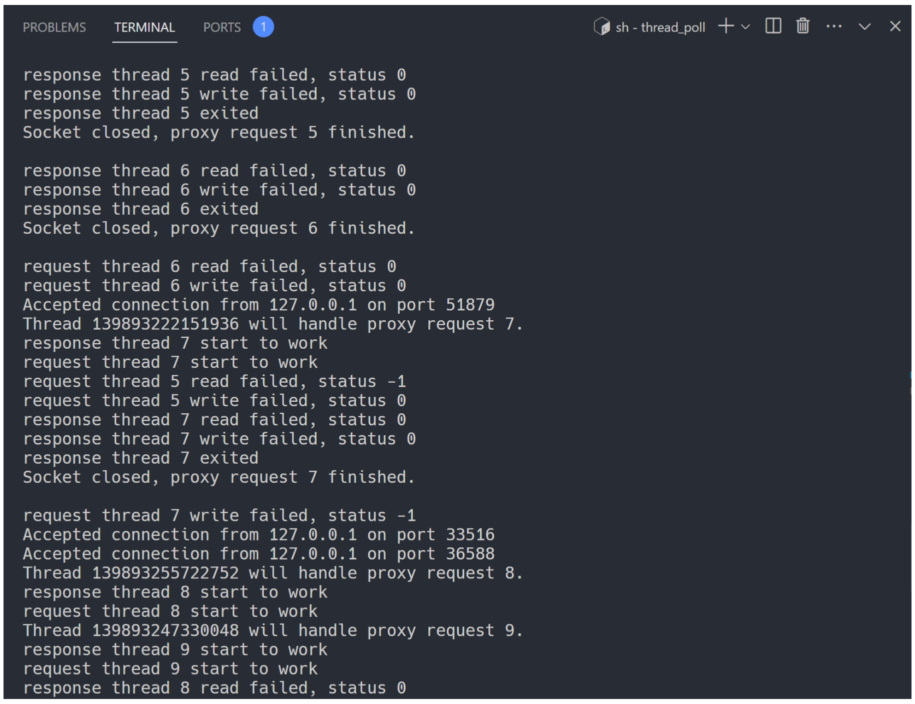
```
vagrant@csc3150:~/csc3150/Assignment_2_121090642/3150-p2-bonus-main/thread_poll$ sh _test
.sh
Listening on port 8000...
Accepted connection from 127.0.0.1 on port 41180
Thread 139893238937344 will handle proxy request 0.
response thread 0 start to work
request thread 0 start to work
response thread 0 read failed, status 0
response thread 0 write failed, status 0
response thread 0 exited
Socket closed, proxy request 0 finished.

request thread 0 read failed, status 0
request thread 0 write failed, status 0
Accepted connection from 127.0.0.1 on port 29933
Thread 139893230544640 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.

Accepted connection from 127.0.0.1 on port 30957
Thread 139893222151936 will handle proxy request 2.
response thread 2 start to work
request thread 2 start to work
request thread 1 read failed, status -1
request thread 1 write failed, status 0
response thread 2 read failed, status 0
response thread 2 write failed, status 0
response thread 2 exited
Socket closed, proxy request 2 finished.
```

Figure 12: Terminal output - 1



```
request thread 2 read failed, status 0
request thread 2 write failed, status 0
Accepted connection from 127.0.0.1 on port 57564
Accepted connection from 127.0.0.1 on port 58076
Thread 139893255722752 will handle proxy request 3.
response thread 3 start to work
request thread 3 start to work
Thread 139893247330048 will handle proxy request 4.
request thread 4 start to work
response thread 4 start to work
response thread 3 read failed, status 0
response thread 3 write failed, status 0
response thread 3 exited
Socket closed, proxy request 3 finished.

response thread 4 read failed, status 0
response thread 4 write failed, status 0
response thread 4 exited
Socket closed, proxy request 4 finished.

request thread 4 read failed, status 0
request thread 4 write failed, status 0
Accepted connection from 127.0.0.1 on port 13973
Thread 139893238937344 will handle proxy request 5.
request thread 5 start to work
response thread 5 start to work
request thread 3 write failed, status -1
Accepted connection from 127.0.0.1 on port 18069
Thread 139893230544640 will handle proxy request 6.
response thread 6 start to work
request thread 6 start to work
response thread 5 read failed, status 0
```

Figure 13: Terminal output - 2



```
response thread 5 read failed, status 0
response thread 5 write failed, status 0
response thread 5 exited
Socket closed, proxy request 5 finished.

response thread 6 read failed, status 0
response thread 6 write failed, status 0
response thread 6 exited
Socket closed, proxy request 6 finished.

request thread 6 read failed, status 0
request thread 6 write failed, status 0
Accepted connection from 127.0.0.1 on port 51879
Thread 139893222151936 will handle proxy request 7.
response thread 7 start to work
request thread 7 start to work
request thread 5 read failed, status -1
request thread 5 write failed, status 0
response thread 7 read failed, status 0
response thread 7 write failed, status 0
response thread 7 exited
Socket closed, proxy request 7 finished.

request thread 7 write failed, status -1
Accepted connection from 127.0.0.1 on port 33516
Accepted connection from 127.0.0.1 on port 36588
Thread 139893255722752 will handle proxy request 8.
response thread 8 start to work
request thread 8 start to work
Thread 139893247330048 will handle proxy request 9.
response thread 9 start to work
request thread 9 start to work
response thread 8 read failed, status 0
```

Figure 14: Terminal output - 3

4 Learn from the task

From the task, I have learnt something below:

1. How to compile a cpp file with multiple pthreads in Linux command line.
2. How to use `printf()` function to control the cursor on the terminal. For example, `printf("\033[2J")` is used to clean the terminal. This is important to create a mini game based on cpp on terminal.
3. How to create multiple pthreads. How to lock the mutex, and unlock them to protect the shared space. How to exit each pthread for safe termination.
4. From the bonus task, I learn further more about the pthreads and master the function of the condition variables to control the threads.
5. I also learn how to forward a port outside a Linux virtual machine and use the browser to observe the result.