The Chinese University of Hong Kong, Shenzhen

CSC3150

Operating Systems

Assignment2 Report

Author: Yuhang Wang 王禹杭

Student Number ID: 120090246

October 23, 2022

Contents

1. Design

1.1 Task

Assignment 2 requires us to implement a game using multithread programming.

The game allow user to control the frogs to jump between the moving logs, and finally, when the frog crosses the river, you will win the game.

I will explain my ideas in terms of **variables, functions, threads usage, logs moving, program flow, and mutex lock**.

a) Variables: I have set some global variables such as **finish**, **state**, and **direction** to represent whether the game finishes, the state of the game, and the direction of the logs that the frogs stand, respectively. Also, there is a global **map** which stores the map of the game, and the map will be printed in the terminal.

b) Functions: "kbhit" function is to determine whether the keyboard is hit or not. It will return 1 if yes, and return 0 if not. "Logs_move" is the most important of the functions which controls the movement of the logs. "frogs_move" is a function which calls "kbhit" function until the game finishes, and move the frogs according to the instruction that user inputted. Also, there is a function called "print_map" which updates the map printed in the terminal and finally print the state: win, lose, or quit.

c) Thread usage: I created three threads to run concurrently. One is for printing the map, another one is for updating the positions of the logs, the other is for detecting the user's input. They should run simultaneously with following the rules of mutex locking.

d) Mutex locking: Mutex lock is the primary method to implement thread synchronization and protect shared data. When a piece of the code is locked, the threads cannot perform the operations simultaneously. Multiple writes and multiple reads are not allowed. The lock provides the protection of accessing the shared data resource. As a result, it should be added to the processes which deal with the shared data (global variables). I firstly initialize a mutex lock in my main function, and then in each thread, I put the part that accesses the shared data between the "pthread_mutex_lock" and "pthread_mutex_unlock". It is noticeable that I do unlock before using "usleep" function.

e) Logs moving: Each time the game starts, I use "srand" and "rand" to generate a random initial position of the logs. In each iteration until the game finishes, I change the initial position by 1 horizontally and update the logs using "=" on the map, and frog will also change its position according to the value of "direction". In the end, I use **usleep(100000)** to control the speed of the logs moving, the less the sleeping time, the more the speed.

f) Program Flow: At first, the map will be initialized, and the frog will stand at one side of the river. Then, initialize the pthread and mutex lock, and create 3 threads (log, frog, print). Then join then together, then destroy the mutex lock, etc.

1.2 Bonus

The key idea: In the thread pool programming mode, the task is submitted to the entire thread pool, not directly to a thread. After the thread pool gets the task, it searches internally to see if there are any idle threads, and if so, submit the task to the thread pool. to an idle thread. And a thread can only execute one task at a time, but can submit multiple tasks to a thread pool at the same time. Thus, it greatly saves the resources of the computer.

In "async_init" function, I initialize the thread pool, the queue, the mutex lock, and the conditional signal. And I call my function "worker" in the "pthread_create" function.

My "worker" function will do the "pthread_cond_wait" function until the size of the queue is 0, and it will also get the function name and the arguments form the head of the queue, then it will execute the function got.

In "async_run" function, I rewrite it to support the thread pool, and I use "DL_APPEND" to add a task to my queue, and it will call "pthread_cond_signal" to signal a signal.

2 Environment and Execution

2.1 Environment

a) **Version of Linux Distribution:**

Distributor ID: Ubuntu

Description: Ubuntu 16.04.7 LTS

Release: 16.04

Codename: xenial

b) **Version of kernel: 5.10.99**

c) **Version of gcc: 5.4.0 20160609**

2.2 Execute My Program

2.2.1 Task

After opening the terminal, just cd to the source folder and type:

g++ hw2.cpp -lpthread

./a.out

Then you can play the game.

```
vagrant@csc3150:~/csc3150/Assignment_2_120090246/source$ g++ hw2.cpp -lpthread
vagrant@csc3150:~/csc3150/Assignment_2_120090246/source$ ./a.out
```

2.2.2 Bonus

After opening the terminal, just cd to the source folder and type:

./httpserver --files files/ --port 8000 --num-threads T

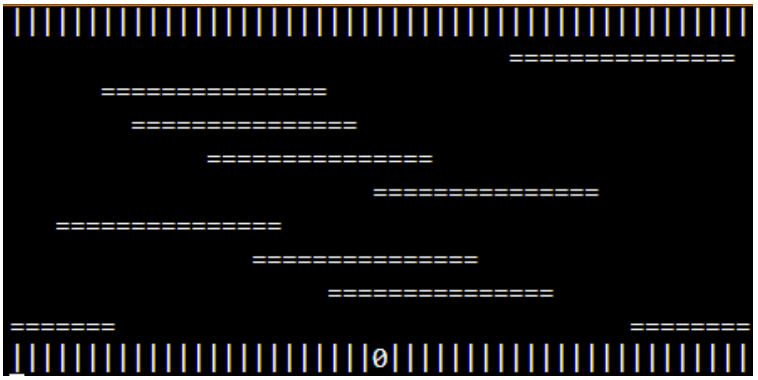
(Please replace T with a **thread number**, say 10) Then you open another terminal and type:

ab -n X -c T http://localhost:8000/

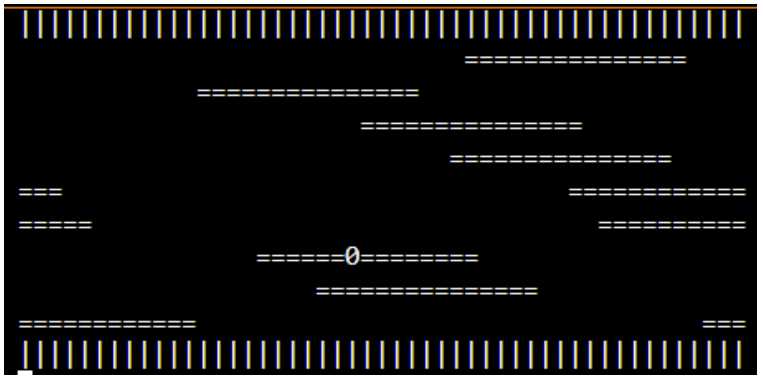
(Please replace X with the total **request number**, say 5000, and T with the thread **number**, say 10). Then you can see outputs in both the terminals.

3 Task Outputs

3.1 Start:



3.2 Playing:



3.3 Win:

```
You win the game!!
vagrant@csc3150:~/csc3150/Assignment_2_120090246/source$
```

3.4 Lose:

```
You lose the game!!
vagrant@csc3150:~/csc3150/Assignment_2_120090246/source$
```

3.5 Quit:

```
You exit the game.
vagrant@csc3150:~/csc3150/Assignment_2_120090246/source$
```

4 Bonus Outputs

```
vagrant@csc3150:~/csc3150/Assignment_2_120090246/3150-p2-bonus-main/thread_poll$ ab -n 5000 -c 10 http://localhost:800
This is ApacheBench, Version 2.3 <$Revision: 1706008 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)
Completed 500 requests
Completed 1000 requests
Completed 1500 requests
Completed 2000 requests
Completed 2500 requests
Completed 3000 requests
Completed 3500 requests
Completed 4000 requests
Completed 4500 requests
Completed 5000 requests
Finished 5000 requests


Server Software:
Server Hostname:        localhost
Server Port:            8000

Document Path:          /
Document Length:        4626 bytes

Concurrency Level:      10
Time taken for tests:   1.681 seconds
Complete requests:      5000
Failed requests:        0
Total transferred:      23460000 bytes
HTML transferred:       23130000 bytes
Requests per second:    2974.01 [#/sec] (mean)
Time per request:       3.362 [ms] (mean)
Time per request:       0.336 [ms] (mean, across all concurrent requests)
Transfer rate:          13627.00 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   0.5      1       7
Processing:     0    2  13.4      1     173
Waiting:        0    2  13.4      1     173
Total:          0    3  13.3      2     173

Percentage of the requests served within a certain time (ms)
  50%      2
  66%      2
  75%      3
  80%      3
  90%      3
  95%      4
  98%      5
  99%     10
 100%    173 (longest request)
```

5 Learnings

5.1 To write multi-thread programs.

Based on the lectures and the tutorials, I have a deeper understanding about the creation, termination, join, signal conditions and mutex lock of the threads. And this programs let me know more about the advantages of the multi-thread programming, and it also makes me more interested about the multi-thread programming.

5.2 To divide the whole project into parts

The core of multi-thread is to divide the task into different parts and solve them separately. In the task, I simply divide the game to three parts to run concurrently. And in each part, I can just try to achieve the current goal.

5.3 To get practical knowledge

The use of while and “usleep()” can make the interface dynamic, just like animation. The macro definition “#define” can make the code clearer. The "\033[H” can clear the terminal and move the cursor. Also, I learned about compile the cpp and c file in Linux, the idea to design some simple games, the methods for debugging, and the way to encapsulate processes using functions.

5.4 From bonus

I learned to implement a thread pool my myself. Which definitely deepen my understanding about the multithread programming.

What’s more, without the thread pool, the system continuously starts and closes new threads, which is very expensive, and will consume system resources excessively, and the danger of switching threads over time may lead to the collapse of system resources. At this time, the thread pool is the best choice.