Assignment2 Report

Name:余永琦 ID:120090761

*Note: I have finished the bonus.

How did I design my program?

In this assignment, we are asked to write a multi-thread program to implement a game called “Frog crosses river”. It is a game to let the player control a frog, and make it cross the river successfully. If the frog falls into water or cross boundaries, the player loses. If the frog reaches the other river bank, the player wins.

To design the game, we mainly need to do the following things: keep the logs moving, capturing the user input, and also print the map in terminal. So, we can design three threads to do the above three jobs.

In logs moving, we moves the logs in the map. If the frog is in a log, then we need to move the whole log. Otherwise we can only move the starting piece and ending piece of the logs to get it moved. Also, we need to check if the frog cross the boundaries as the logs move, if so, the player will lose the game. Finally, set an appropriate frequency for the log move using “usleep()” function.

In capturing keyboard inputs, we keep listening on the keyboard and check whether the frog falls into the river or cross boundaries or reaches the other river bank.

In map printing, we just need to print all elements we store in the map, clear the terminal every time, set an appropriate frequency as well.

I defined a local variable status to indicate the status of the game (0 for normal executing, 1 for winning, 2 for losing and 3 for quitting). Also, I declared a mutex lock so that when a thread is running, we can make sure no other thread can change the shared source.

In the main function, we need to initialize the mutex variable, initialize the map and log, create 3 threads and then join them. Finally we destroy the mutex and output the result according to the variable status. Now we finish the job!

Bonus

In bonus, we are asked to implement correctly two important functions of a thread pool, one is to initialize the thread pool, the other is to run the input function by the threads in the thread pool.

First, I defined a queue to store all the tasks, each node is consisted of a function pointer, a int type argument, and a pointer pointing to next node. Then the queue can be defined by only its head (the first node). Also, a mutex and conditional variable is declared to be a global variable.

Then, I defined a work function for each thread, for each thread, I let it go to wait until it receive signal. Then the thread wakes up and do the task in the task queue. This sleep and wait status changing is implemented by mutex and conditional variable. Each time when there is a task, we wake some threads up and let it finish the task.

For thread pool initialization, we need to create a number of threads according to the argument, and set all start-routine to be the work function we defined before.

Finally, in the “async_run” function, we create a task node, passing all the arguments into the node, and then insert the node to our task queue. Then, we send signal to our thread pool, the task will be done.

For the implementation details, please check my code and read the comments.

Develop Environment

OS: Ubuntu 16.04.7 LTS

Kernel: 5.10.146

gcc: 5.4.0

Steps to execute my program

Frog Game

1. $ make
2. $ ./a.out

Bonus

1. $ make
2. $ ./httpserver --proxy inst.eecs.berkeley.edu:80 --port 8000 --num-threads

Note: Step 2 of bonus is to open a server and test if our thread pool works. In addition, you can use "$ make clean" to clear the binary object file.

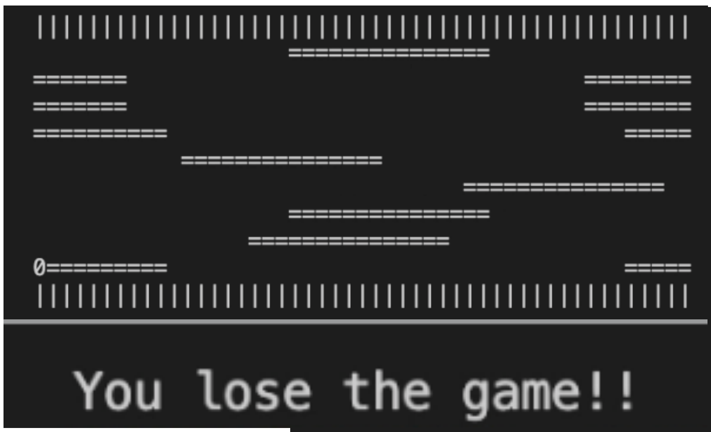
What did I learn in the tasks?

In the frog game, I learned how to do multiple threads. Nowadays, the multi-thread program is almost everywhere, because it is very efficient. By designing this game, I learned how to depart the function of a program into many part, and use multiple threads to handle each part. Also, I learned that the share space should be locked when a thread is accessing the data, otherwise other threads may change the value during this thread executing, and we could get an unexpected result. Hence, coordinating the threads is a very important task that a programer need to handle.

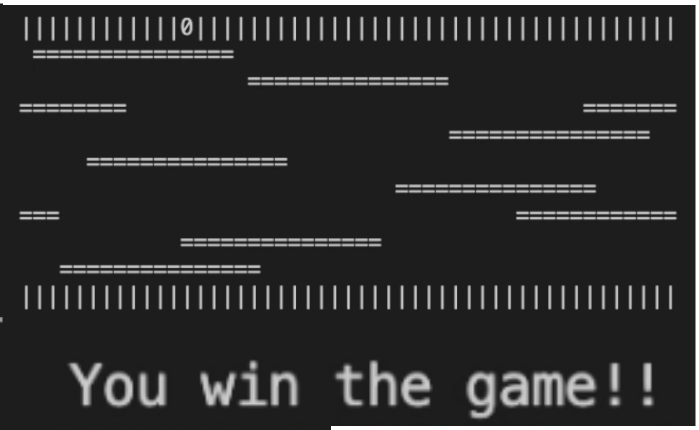
In bonus, I learned how a thread pool works. The thread pool is widely applied in heavy programs. In this task, I get familiar with the status of a thread in thread pool, also when and how to change its status. More detailed, I knew that a thread show go to wait status instead of ready status to avoid some costing task such as IO accessing always stays in the ready status and repeatedly cost lots of time. Also, I learned using the mutex and conditional variable together to change the threads, let a thread go to wait or wake a the thread up.

Screen Shot

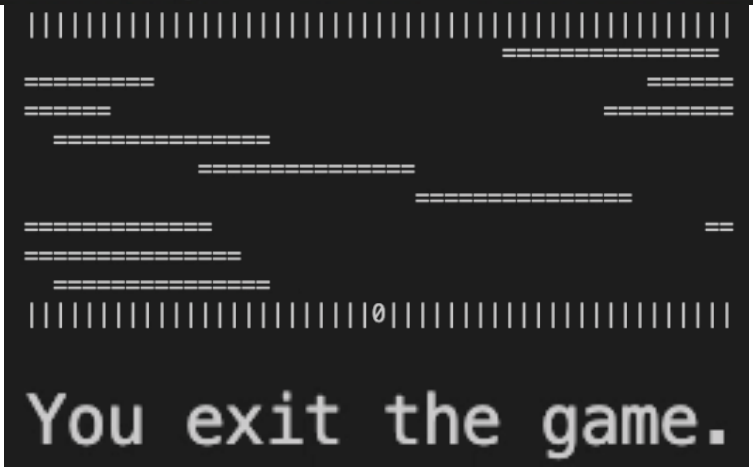
Frog Game



```
You lose the game!!
```

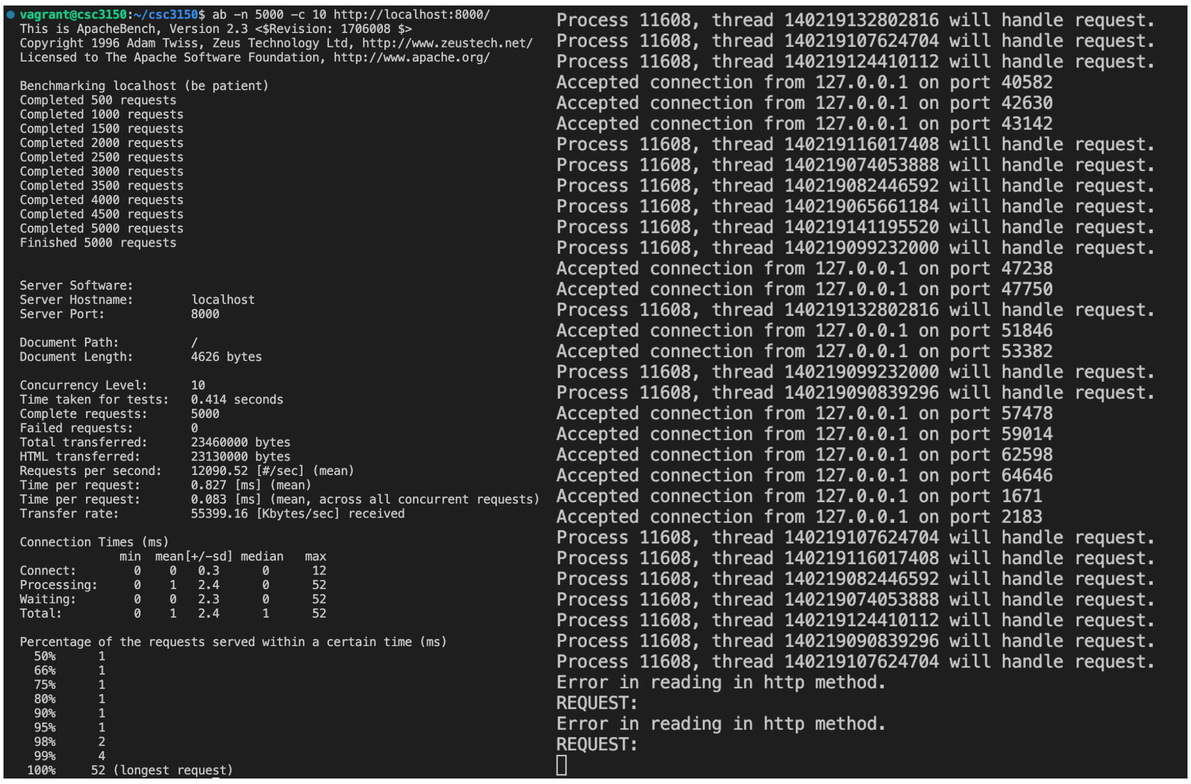


```
You win the game!!
```



```
You exit the game.
```

Bonus



```
vagrant@csc3150:~/csc3150$ ab -n 5000 -c 10 http://localhost:8000/
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:   0.414 seconds
Complete requests:      5000
Failed requests:        0
Total transferred:      23460000 bytes
HTML transferred:       23130000 bytes
Requests per second:    12090.52 [#/sec] (mean)
Time per request:       0.827 [ms] (mean)
Time per request:       0.083 [ms] (mean, across all concurrent requests)
Transfer rate:          55399.16 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.3      0      12
Processing:     0    1   2.4      0      52
Waiting:        0    0   2.3      0      52
Total:          0    1   2.4      1      52

Percentage of the requests served within a certain time (ms)
  50%      1
  66%      1
  75%      1
  80%      1
  90%      1
  95%      1
  98%      2
  99%      4
 100%     52 (longest request)
```

```
Process 11608, thread 140219132802816 will handle request.
Process 11608, thread 140219107624704 will handle request.
Process 11608, thread 140219124410112 will handle request.
Accepted connection from 127.0.0.1 on port 40582
Accepted connection from 127.0.0.1 on port 42630
Accepted connection from 127.0.0.1 on port 43142
Process 11608, thread 140219116017408 will handle request.
Process 11608, thread 140219074053888 will handle request.
Process 11608, thread 140219082446592 will handle request.
Process 11608, thread 140219065661184 will handle request.
Process 11608, thread 140219141195520 will handle request.
Process 11608, thread 140219099232000 will handle request.
Accepted connection from 127.0.0.1 on port 47238
Accepted connection from 127.0.0.1 on port 47750
Process 11608, thread 140219132802816 will handle request.
Accepted connection from 127.0.0.1 on port 51846
Accepted connection from 127.0.0.1 on port 53382
Process 11608, thread 140219099232000 will handle request.
Process 11608, thread 140219090839296 will handle request.
Accepted connection from 127.0.0.1 on port 57478
Accepted connection from 127.0.0.1 on port 59014
Accepted connection from 127.0.0.1 on port 62598
Accepted connection from 127.0.0.1 on port 64646
Accepted connection from 127.0.0.1 on port 1671
Accepted connection from 127.0.0.1 on port 2183
Process 11608, thread 140219107624704 will handle request.
Process 11608, thread 140219116017408 will handle request.
Process 11608, thread 140219082446592 will handle request.
Process 11608, thread 140219074053888 will handle request.
Process 11608, thread 140219124410112 will handle request.
Process 11608, thread 140219090839296 will handle request.
Process 11608, thread 140219107624704 will handle request.
Error in reading in http method.
REQUEST:
Error in reading in http method.
REQUEST:
```