Report

Zhenyu Pan 120090196

October 23, 2022

Contents

1 Frog Game

1.1 Design of the program

1.1.1 Basic idea

The main idea of the program is to implement the game "Frog crosses river" using multi-thread programming. The core of my program is to create **9 threads** to control the move of 9 logs using function *phread_create()*. Notice that each thread points to function *logs_move()* which has four usages: first, it makes the log move randomly; second, it can be activated by keyboard hit and the frog will move correspondingly; third, it will detect the current situation of the game (win, lose, or quit); fourth, it updates the current map and prints it out. After all the threads terminates, print out the result of the game according to the status received.

1.1.2 main() function

The main function first use the given code to create a static river map and set frog at the starting position. After that, initialise the mutex referenced by *mutex* with default attribute using function *pthread_mutex_init()*. At the same time, check whether the mutex is initialised successfully by checking the return value. Then declare 9 threads using *pthread_t* and create them using function *pthread_create()*. All of them point to the function *logs_move()* which we will focus on later. To wait for all the 9 threads terminates normally(also means game over), use function *pthread_join()* to make it. In the end, when all the threads terminates, first clear the window, and print out the corresponding information(exist, win or lose). One more attention, don't forget to destroy the mutex.

1.1.3 logs_move() function

First use the *srand(time(NULL))* function to set the birth position of logs randomly. Then use function *rand()* to set the length of logs randomly(from 10-15). After that, enter the while loop to move the logs and frog until game over. In the while loop, first set appropriate parameters to function *usleep()* to make the logs "move". Then lock the mutex using function *pthread_mutex_lock*, which is used for ensuring that only one thread can change the map at one

single instant, and only one thread can do the *kbhit()* at a particular instant. After each log execute once in the while loop, the variable "count" will plus 1, after all the 9 threads execute one time, clean the window and update the map. Repeat the above process until the flag "stop" is true. For the frog movement, using function *kbhit()* to detect the input and do the corresponding work. When *kbhit()* is activated, determine whether the frog is dead. If the frog jumps into the river, lose the game. If the frog is still alive after movement, modify the map. When the frog is not activated, make it move with the log. (More implementation details pls check the comments in source code). Notice that after each execution in while loop, remember to unlock the mutex. Lastly, when the flag "stop" is true, jump out of the while loop.

1.2 Environment of running the program

```
vagrant@csc3150:~/csc3150/Assignment_2_120090196$ cat /etc/issue
Ubuntu 16.04.7 LTS \n \l
```

Figure 1: OS version

```
vagrant@csc3150:~/csc3150/Assignment_2_120090196$ uname -r
5.10.5
```

Figure 2: Kernel version

```
vagrant@csc3150:~/csc3150/Assignment_2_120090196$ gcc --version
gcc (Ubuntu 5.4.0-6ubuntu1~16.04.12) 5.4.0 20160609
```

Figure 3: gcc version

1.3 Steps to execute the program

First, type "sudo make" on the terminal. Then, type "./a.out".

```
vagrant@csc3150:~/csc3150/Assignment_2_120090196/source$ sudo make
g++ -std=c++11  hw2.cpp  -o a.out -lpthread
vagrant@csc3150:~/csc3150/Assignment_2_120090196/source$ ./a.out
```

Figure 4: Steps to execute

1.4 Screenshot of program output

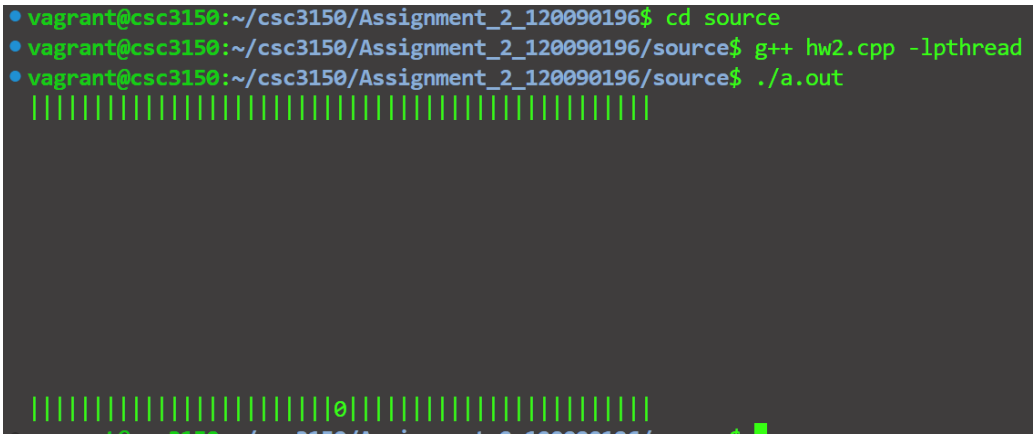

Figure 5: Template static output

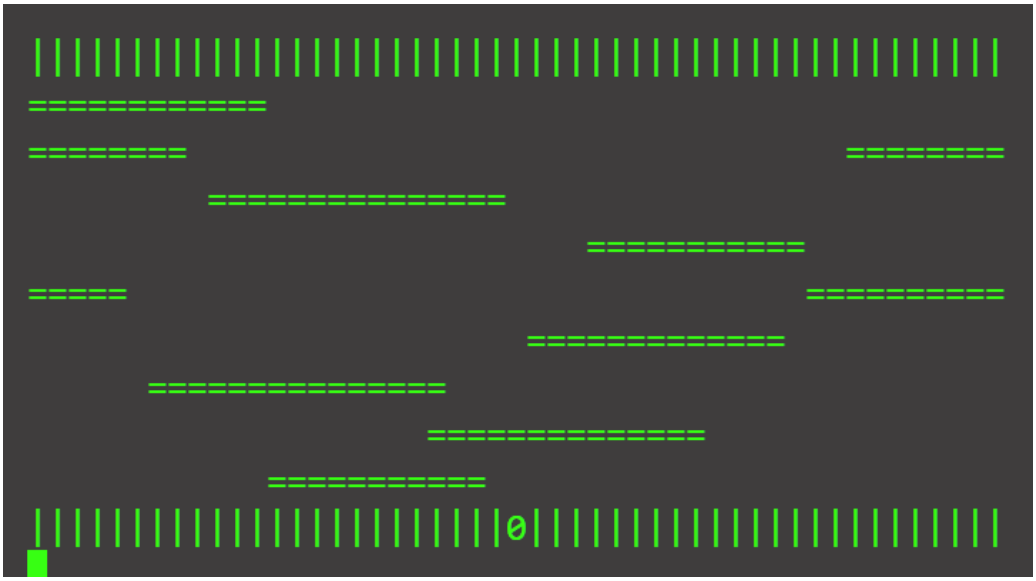

Figure 6: Start

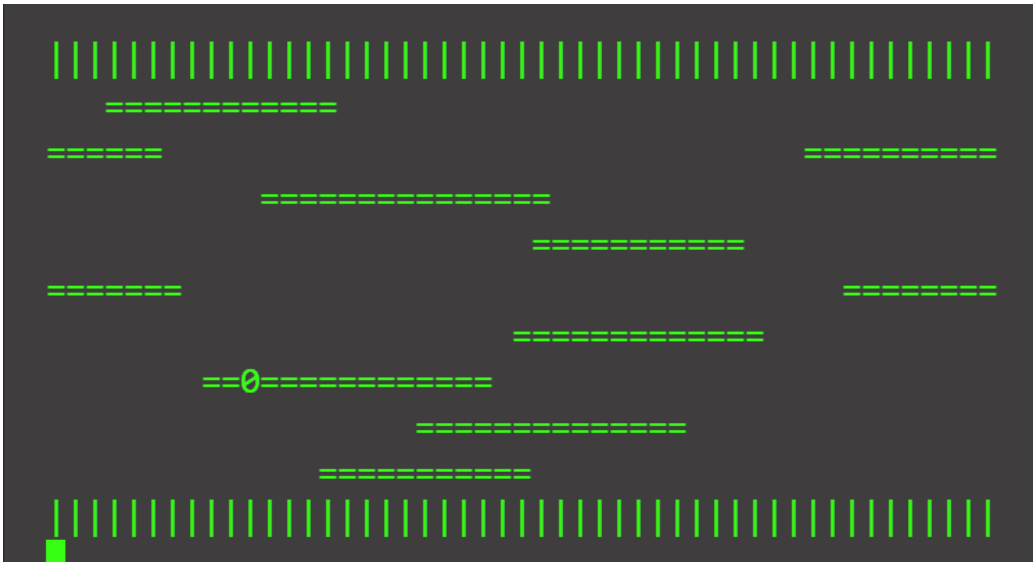

Figure 7: Process

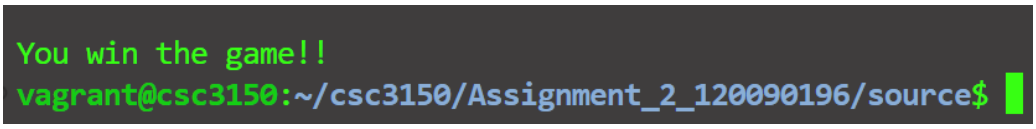

Figure 8: Win

Figure 9: Lose

Figure 10: Exit

1.5 What I learnt

1. Listen to tutorial carefully
2. Have a deeper understanding of multi-thread programming
3. Practice C/C++ skills

2 Bonus

2.1 Design of the program

The main idea of the program is to implement a thread pool(I think is similar to the concept of buffers in OS), which is used to prepare threads in advance and reuse threads. To sum up, create special threads whose task is to execute other functions from a task queue. These threads "sleep" when there is no task, and wake up(sem_post) when there is a task to execute. Note that for *async_run()*, as required, it will return immediately before the job is handled(only responsible for putting the task to the task queue, no care about who execute it). More details please check the source code and the comments in it.

2.2 Screenshot of program output

```
vagrant@csc3150:~/csc3150/Assignment_2_120090196/3150-p2-bonus-main/thread_poll$ 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:   0.489 seconds
Complete requests:      5000
Failed requests:        0
Total transferred:      23460000 bytes
HTML transferred:       23130000 bytes
Requests per second:    10226.85 [#/sec] (mean)
Time per request:       0.978 [ms] (mean)
Time per request:       0.098 [ms] (mean, across all concurrent requests)
Transfer rate:          46859.76 [Kbytes/sec] received
```

Figure 11: Test on piazza

```
Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.2      0       3
Processing:     0    1   1.8      0      39
Waiting:        0    1   1.8      0      39
Total:          0    1   1.8      1      39

Percentage of the requests served within a certain time (ms)
  50%      1
  66%      1
  75%      1
  80%      1
  90%      1
  95%      2
  98%      2
  99%      4
 100%     39 (longest request)
```

Figure 12: Test on piazza

```
vagrant@csc3150:~/csc3150/Assignment_2_120090196/3150-p2-bonus-main/thread_poll$ ./httpserver --proxy inst.eecs.berkeley.edu:80 --port 8
000 --num-threads 5
create new thread 1
create new thread 2
create new thread 3
create new thread 4
create new thread 5
Listening on port 8000...
Accepted connection from 127.0.0.1 on port 6794
Accepted connection from 127.0.0.1 on port 7306
Thread 140298964084480 will handle proxy request 0.
Thread 140298955691776 will handle proxy request 1.
response thread 0 exited
request thread 0 exited
Socket closed, proxy request 0 finished.

Accepted connection from 127.0.0.1 on port 9354
Thread 140298947299072 will handle proxy request 2.
Accepted connection from 127.0.0.1 on port 11402
Accepted connection from 127.0.0.1 on port 11914
Thread 140298856425216 will handle proxy request 3.
Thread 140298864817920 will handle proxy request 4.
response thread 1 exited
response thread 3 exited
response thread 4 exited
response thread 2 exited
```

Figure 13: Test on README.md

2.3 What I learnt

1. Self study is important
2. Self study is important
3. Self study is important