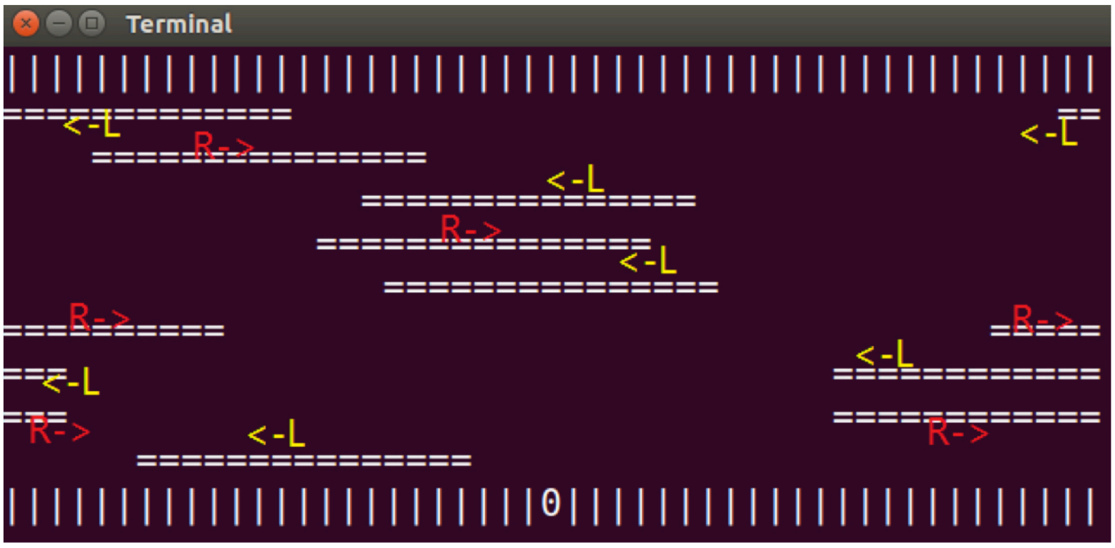
Report for CSC3150 Assignment 2

Yang Liang　　120090874

1. Program Design Methodology

1.1 Program Task

In this assignment, we are required to design a game “Frog Crosses the River” by the use of multithread programming techniques. In the game, there is a river with logs floating on it, and a frog is standing in the middle of the bottom bank of the river. It must cross the river by jumping on the logs as they pass by staggerly. The user controls the frogs’ moving directions *(upward, downward, left, right)* by keyboard, and will win if the frog jumps to the other bank of the river successfully, lose if the frog falls into the river, or the frog reaches the left/right of the river while still staying on the log. In actual implementation, the river banks will be represented by the symbols “||||||||||||||||”, the frog is represented by “*0*”, and the logs are as “= = = =”, as shown below.



1.2 Implementation Details

In the actual design, we mainly need to tackle 3 aspects. One for handling the user commands of the frog’s motion, one for the stagger movements of the logs, and the other for dynamically displaying the game interface to player.

Totally 11 threads are created for the implementation, including 9 threads for tracing the movements of each log in the river, respectively (executing the function *"logs_move()"*); 1 thread for tracing the movement of the frog (executing the function *"frog_move()"*), and another 1 thread for displaying the screen (executing the function *"render_screen ()"*). The shared data (critical section) among the thread includes variable "*stop_process*", which controls whether to terminate the execution of the game and display the final result; *"flag"*, which controls the termination status (1 for user "quit", 2 for "win", 3 for "lose"); and "*map"*, a 2-dimensional array that stores the information for displaying the game. In order to protect these shared data when multiple writes and modifying occur, a mutex lock *"mutex"* is created and used in my defined function *"frog_move()"*, *"logs_move()"* and *"render_screen()"*, which will be introduced one-by-one as follows.

Function *"frog_move()"* is implemented to trace the movement of the frog according to the keyboard input of users. Function *getchar()* and *kbhit()* is used to get the input moving direction from user keyboard. Changes will be described subsequently, for example, moving upwards means the "*row"* coordinate of the frog should be decreased by 1, i.e., when shown in the *map*, it is *map[frog.x – 1][frog.y] = '0'*, and its original position should be replaced by log or river bank, i.e., *map[frog.x][frog.y] = '|'*. Of course, the position change is valid only when there is log above it, else the frog will fall into river and fail the game, so we need to validate the movement. The termination of the thread is organized by variable "*stop_process"* in the loop "*while(!stop_process)"*, when the frog reaches the other bank (win) or do some invalid move that will lead it fall to the river (lose), the *"stop_process"* variable will be set to 1 so as to end the thread.

Function *"logs_move()"* is designed to coordinate the movement of the logs. Note that, logs in odd number rows move from left to right, while in even number rows logs move from right to left. Firstly, we need to specify the *start* position of each log. To make it random and start at a relatively graceful position, we may use *srand()* function to set random seeds, and *rand()* function to determine the starting positions of the logs. The following movement of the logs is controlled by the loop *while(!stop_process)*, and there are 2 moving directions. Every iteration corresponds to a single static position of the log and should be displayed by the *render_screen()* function, therefore, the mutex lock should be placed within the iteration body. It is important to refresh the whole row by setting them to empty elements “ ” at the start of each iteration. Worth mentioning, in the row of log with the frog on it, we need to coordinate with the state of the frog by the flag variable *“frog_on_log”*, which indicates whether the frog is exactly on the log (sometimes, the frog may fall out of the log into the river because of the user’s “left” or “right” moving commands), if not, the game will end and the user loses. If the frog is still on the log after the “left” or “right” movements, we need pay attention to changing the frog’s position *frog.y* instantly. For example, when the log is moving leftwards with a frog on it, the *start* position (left handside) of the log should minus one in every iteration, also we need do the same for the frog’s column information *frog.y*. Moreover, in order to make the movements of the log look smooth, the *usleep()* function is adopted to suspend execution of the calling thread, by appropriately adjust the sleeping time interval.

The display of the game interface is completed by the *render_screen()* function. It is trivial since we only need to print out the elements stored in the 2-dimensional array *map* by the *puts()* function. The dynamically display should again be organized in the loop *while(!stop_process),* one print screen in each iteration, with a mutex lock in each iteration and *usleep()* function to display gracefully.

1.3 Bonus

In the bonus task, we are required to implement 2 functions in thread pool, which are *async_init()* and *async_run()*. Thread pool is a way of multitask processing. In a thread pool, there are a certain number of threads, and tasks that need to be processed are arranged in a task queue, which follows the FIFO principal. Every time when there is a thread that is not busy currently, the head of the task queue will be pushed to one of the threads in the thread pool for processing. This way of implementation can maximize the use of threads by keeping them as busy as possible, and can avoid repeatedly creating and destroying threads, which is time-consuming and space-consuming, especially in cases when there are huge number of requests. Asynchronous call means that the caller doesn’t need to wait for the return result from the callee. The workflow for asynchrony may be: the caller sends out a request, after that the caller will get return immediately and can do its own task, the request will be executed asynchronously, the caller will be informed sometimes if the request has been finished.

In the actual implementation, we need to implement a thread pool where requests will be handled right away automatically whenever there is thread that is not busy. According to the principle of the thread pool, we construct 2 data structures, structure *my_item_t* is used to store a tasks list (function that needs to be called), its attributes include the *name of the function that needs to be executed*, and *the function parameters*, and the *next task* of it. The data structure *my_queue_t* is constructed for the thread pool, it includes the information of its *thread_id*, the *task list* that needs to be process, the *mutex lock* and the *conditional signals*. Here we implement 3 functions, *async_init()*, *async_run()* and *execute_routine(),* which acts as an auxiliary function.

The function *async_init()* creates the given number of threads and initializes the above 2 data structures. The function that this thread needs to be executed (the third parameter in the *pthread_create()* function) is *execute_routine()*.

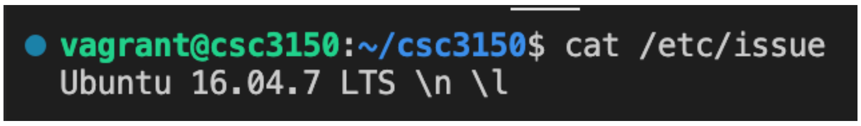
Function *execute_routine()* is the worker function that exactly deals with the tasks. Tasks at the head of the *task list* will be pushed to the thread pool for implementation whenever there is a thread that is not busy. It is organized in the *while(1)* loop to keep the thread executing unless it exits. A mutex lock is placed at the start of iteration to protect share data. When there is no task coming, it goes to sleep by *pthread_cond_wait()* function. When there is work, it will pull out the tasks by sequence from the task list and execute the task according to its *function pointer* and *parameter* attribute.

The function *async_run()* mainly deals with the manipulation of the task list. When there is task coming in, it adds the tasks to the task list *my_item_t*, and will send signal by *pthread_cond_signal()* function to the worker function *execute_routine()*.

Overall, the main workflow is, we use *async_init()* function to create new thread pool and initialize the task list, the threads in the thread pool will execute the worker function *execute_routine(),* which sleeps when there is no tasks, and wakes when new tasks arrive. The *async_run()* function will then handle the task list when new tasks arrive, and send the signal to *execute_routine()* function.

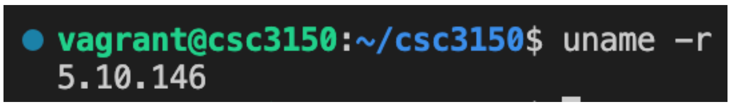
2. Program Execution Environment

2.1 Linux Version



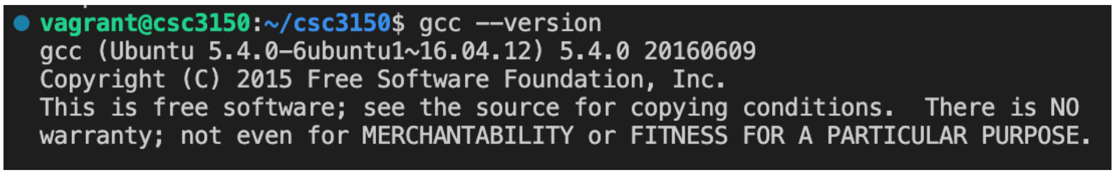
```
vagrant@csc3150:~/csc3150$ cat /etc/issue
Ubuntu 16.04.7 LTS \n \l
```

2.2 Linux Kernel Version



```
vagrant@csc3150:~/csc3150$ uname -r
5.10.146
```

2.3 GCC Version



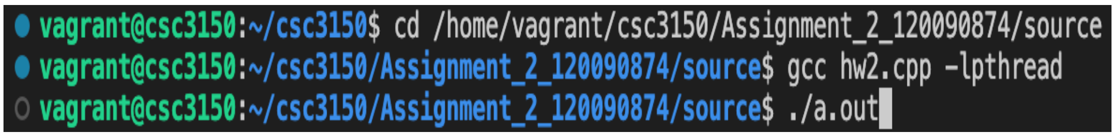
```
vagrant@csc3150:~/csc3150$ gcc --version
gcc (Ubuntu 5.4.0-6ubuntu1~16.04.12) 5.4.0 20160609
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

3. Program Execution Commands

Please use the commands below to execute the program

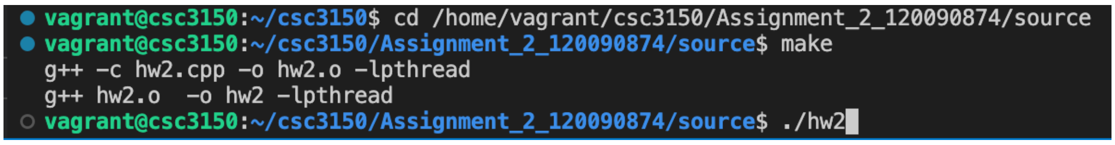
3.1 For the Game (both methods are OK)

Method 1:



```
vagrant@csc3150:~/csc3150$ cd /home/vagrant/csc3150/Assignment_2_120090874/source
vagrant@csc3150:~/csc3150/Assignment_2_120090874/source$ gcc hw2.cpp -lpthread
vagrant@csc3150:~/csc3150/Assignment_2_120090874/source$ ./a.out
```

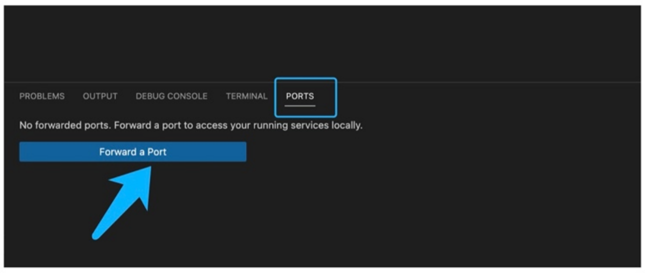
Method 2: (use the Makefile I attached)



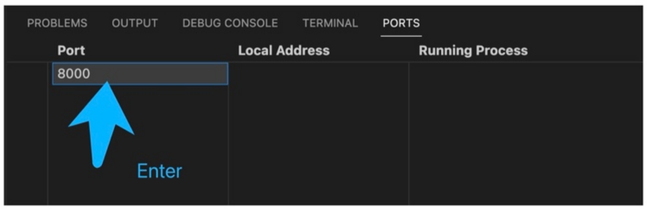
```
vagrant@csc3150:~/csc3150$ cd /home/vagrant/csc3150/Assignment_2_120090874/source
vagrant@csc3150:~/csc3150/Assignment_2_120090874/source$ make
g++ -c hw2.cpp -o hw2.o -lpthread
g++ hw2.o  -o hw2 -lpthread
vagrant@csc3150:~/csc3150/Assignment_2_120090874/source$ ./hw2
```

3.2 For Bonus

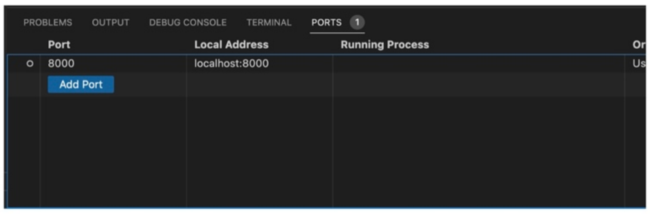
Step1: Set up a port forwarding in VS code as shown below



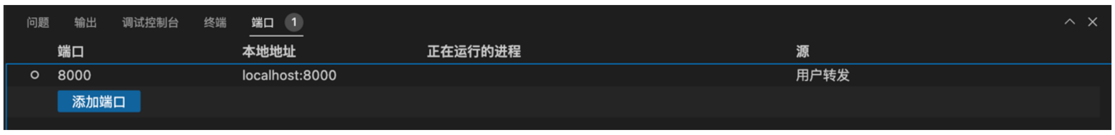
No forwarded ports. Forward a port to access your running services locally.



```
Port          Local Address          Running Process
8000
```

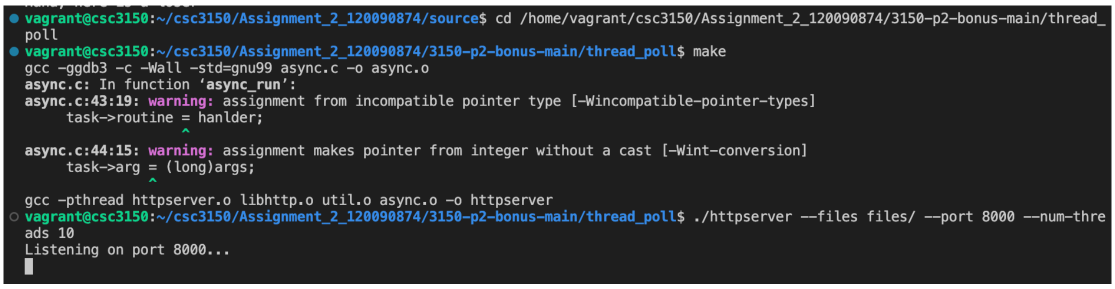


```
Port          Local Address          Running Process          Or
8000          localhost:8000                                  Us
```



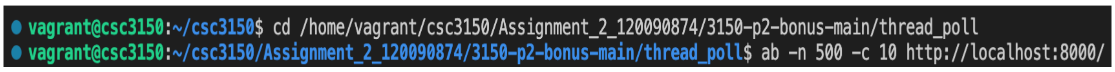
```
端口          本地地址          正在运行的进程          源
8000          localhost:8000                          用户转发
```

Step2: Type the commands as below in one terminal (here we use 10 threads)



```
vagrant@csc3150:~/csc3150/Assignment_2_120090874/source$ cd /home/vagrant/csc3150/Assignment_2_120090874/3150-p2-bonus-main/thread_
poll
vagrant@csc3150:~/csc3150/Assignment_2_120090874/3150-p2-bonus-main/thread_poll$ make
gcc -ggdb3 -c -Wall -std=gnu99 async.c -o async.o
async.c: In function 'async_run':
async.c:43:19: warning: assignment from incompatible pointer type [-Wincompatible-pointer-types]
     task->routine = hanlder;
                   ^
async.c:44:15: warning: assignment makes pointer from integer without a cast [-Wint-conversion]
     task->arg = (long)args;
               ^
gcc -pthread httpserver.o libhttp.o util.o async.o -o httpserver
vagrant@csc3150:~/csc3150/Assignment_2_120090874/3150-p2-bonus-main/thread_poll$ ./httpserver --files files/ --port 8000 --num-thre
ads 10
Listening on port 8000...
```

Step3: In another terminal, use the below command for benchmarking the thread pool implementation (10 threads, 500 requests for example)



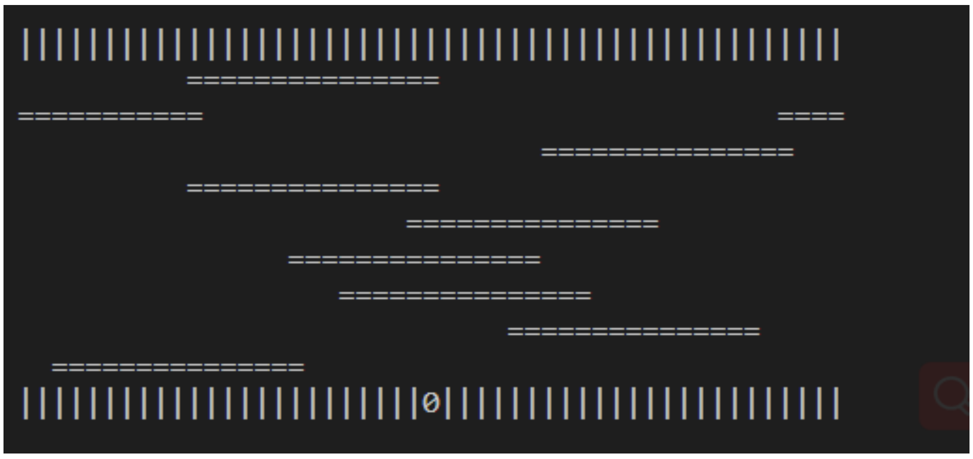
```
vagrant@csc3150:~/csc3150$ cd /home/vagrant/csc3150/Assignment_2_120090874/3150-p2-bonus-main/thread_poll
vagrant@csc3150:~/csc3150/Assignment_2_120090874/3150-p2-bonus-main/thread_poll$ ab -n 500 -c 10 http://localhost:8000/
```

4. What is Learned from this Assignment

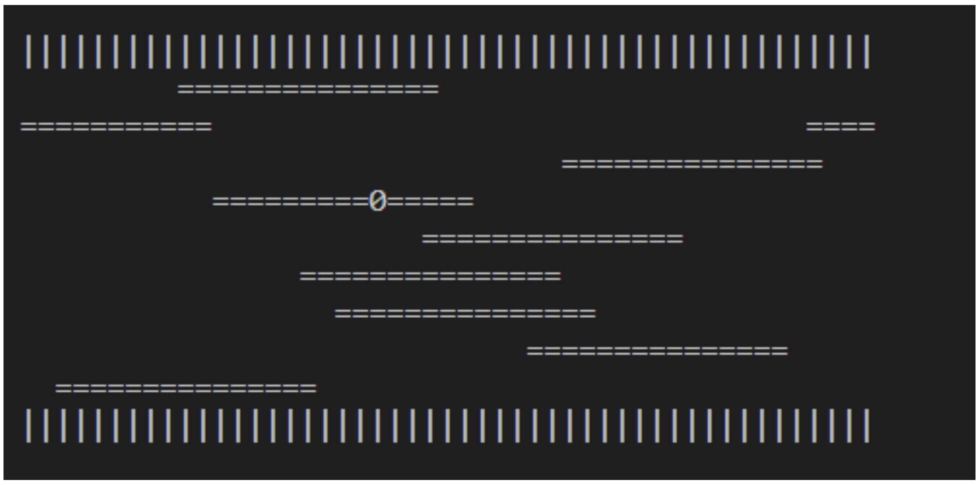
Knowledge of multithread programming is consolidated after the implementation practice of this game. Now I am getting more and more familiar with multithread programming APIs such as *pthread_create()*, *pthread_join()*, and their parameters. Also, in bonus I learned about a practical type of multithread implementation - thread pool. Moreover, by handling and coordinating the threads involved in this program, I have a deeper understanding on the *mutex* mechanism and its protection on reading and writing data in the critical section. Moreover, I get to know some useful functions, such as *kbhit()*, *getchar()* and *puts()* to manage user input, and smoothen the output by appropriately adjusting the time interval in the *usleep()* function. In addition, I am more and more familiar with the C programming language and my debugging skills are improved.

5. Screenshots of the Program Outputs

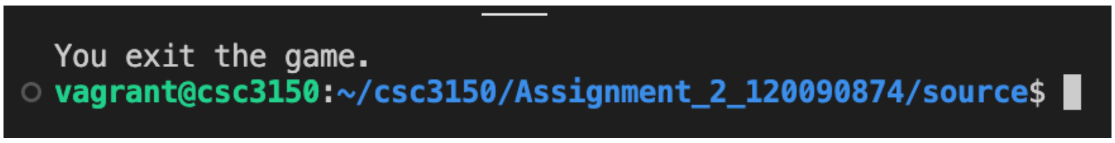
5.1 Output of the Game



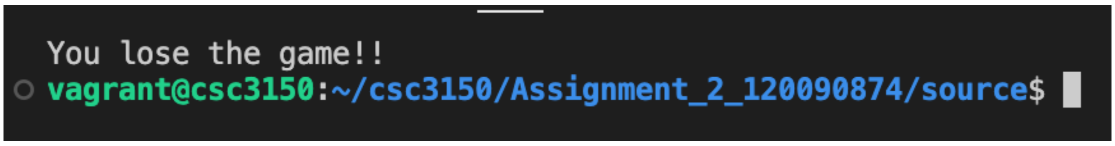
```
|||||||||||||||||||||||||||||||||||||||||||||||||
          ===============
===========                                  ====
                               ===============
          ===============
                       ===============
                ===============
                   ===============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



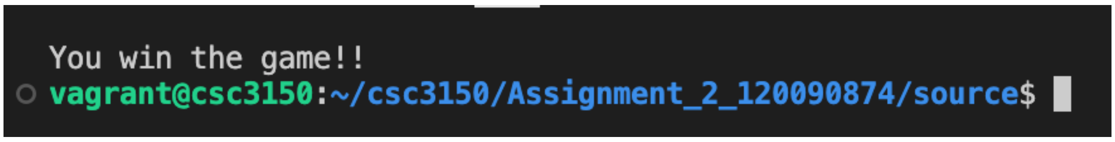
```
|||||||||||||||||||||||||||||||||||||||||||||||||
         ===============
===========                                  ====
                               ===============
           =========0=====
                       ===============
                ===============
                  ===============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



```
You exit the game.
vagrant@csc3150:~/csc3150/Assignment_2_120090874/source$
```



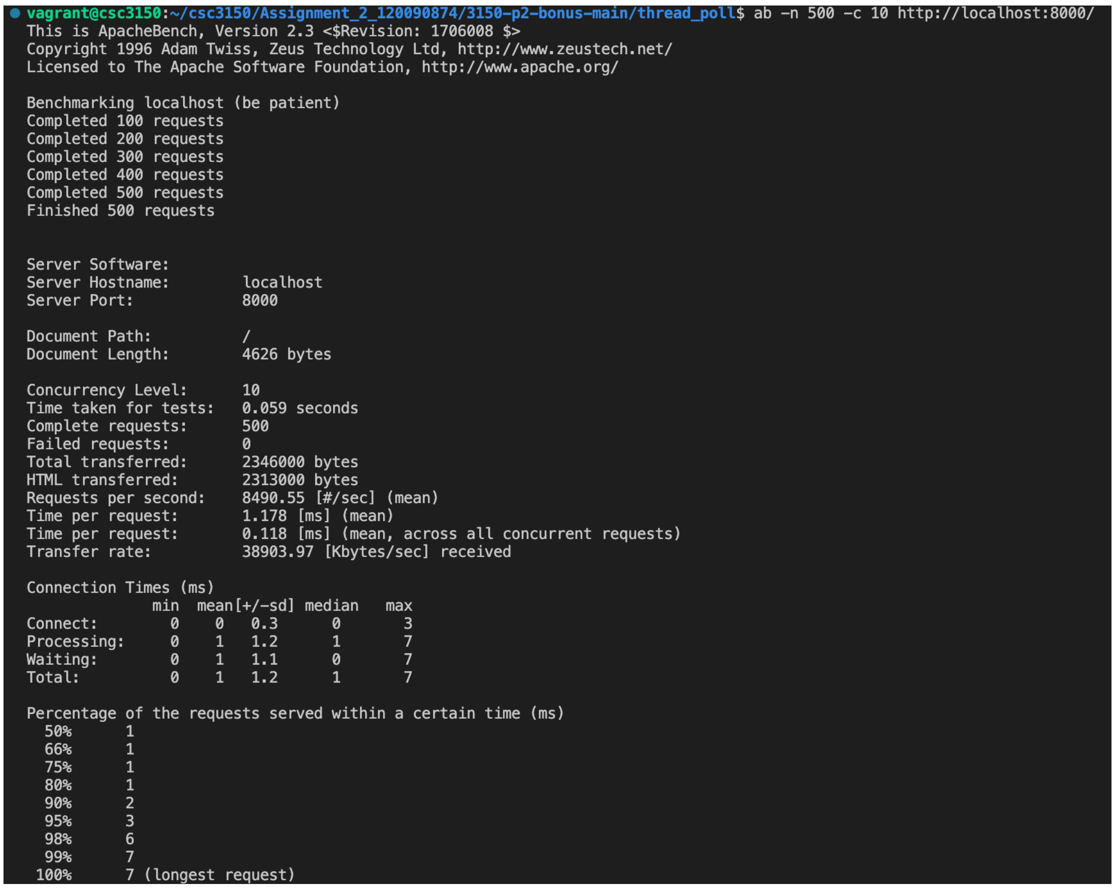
```
  You lose the game!!
○ vagrant@csc3150:~/csc3150/Assignment_2_120090874/source$
```



```
  You win the game!!
○ vagrant@csc3150:~/csc3150/Assignment_2_120090874/source$
```

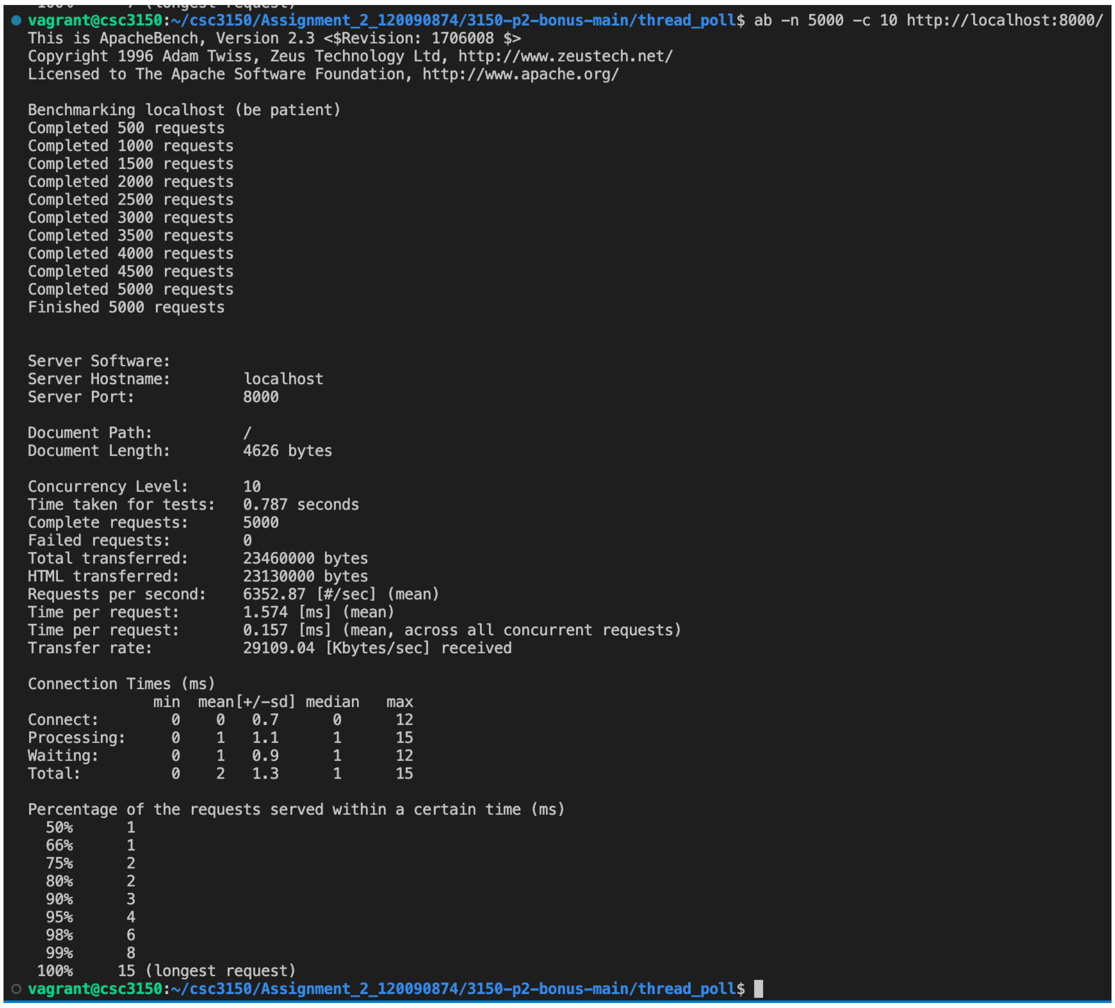
5.2 Output of Bonus

10 threads, 500 requests



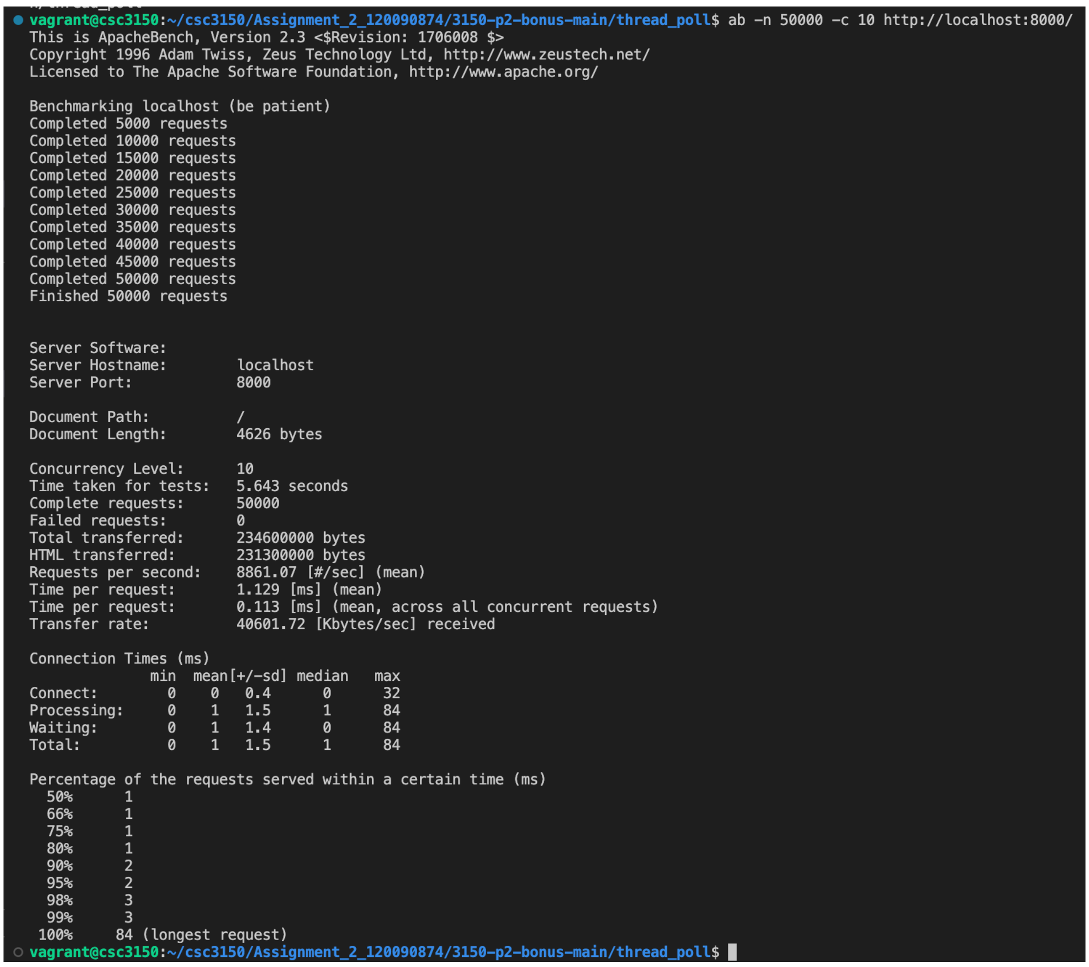
```
● vagrant@csc3150:~/csc3150/Assignment_2_120090874/3150-p2-bonus-main/thread_poll$ ab -n 500 -c 10 http://localhost:8000/
  This is ApacheBench, Version 2.3 <$Revision: 1706008 $>
  Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
  Licensed to The Apache Software Foundation, http://www.apache.org/

  Benchmarking localhost (be patient)
  Completed 100 requests
  Completed 200 requests
  Completed 300 requests
  Completed 400 requests
  Completed 500 requests
  Finished 500 requests


  Server Software:
  Server Hostname:        localhost
  Server Port:            8000

  Document Path:          /
  Document Length:        4626 bytes

  Concurrency Level:      10
  Time taken for tests:   0.059 seconds
  Complete requests:      500
  Failed requests:        0
  Total transferred:      2346000 bytes
  HTML transferred:       2313000 bytes
  Requests per second:    8490.55 [#/sec] (mean)
  Time per request:       1.178 [ms] (mean)
  Time per request:       0.118 [ms] (mean, across all concurrent requests)
  Transfer rate:          38903.97 [Kbytes/sec] received

  Connection Times (ms)
                min  mean[+/-sd] median   max
  Connect:        0    0   0.3      0       3
  Processing:     0    1   1.2      1       7
  Waiting:        0    1   1.1      0       7
  Total:          0    1   1.2      1       7

  Percentage of the requests served within a certain time (ms)
    50%      1
    66%      1
    75%      1
    80%      1
    90%      2
    95%      3
    98%      6
    99%      7
   100%      7 (longest request)
```

10 threads, 5000 requests



```
 100%      7 (longest request)
● vagrant@csc3150:~/csc3150/Assignment_2_120090874/3150-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:   0.787 seconds
  Complete requests:      5000
  Failed requests:        0
  Total transferred:      23460000 bytes
  HTML transferred:       23130000 bytes
  Requests per second:    6352.87 [#/sec] (mean)
  Time per request:       1.574 [ms] (mean)
  Time per request:       0.157 [ms] (mean, across all concurrent requests)
  Transfer rate:          29109.04 [Kbytes/sec] received

  Connection Times (ms)
                min  mean[+/-sd] median   max
  Connect:        0    0   0.7      0      12
  Processing:     0    1   1.1      1      15
  Waiting:        0    1   0.9      1      12
  Total:          0    2   1.3      1      15

  Percentage of the requests served within a certain time (ms)
    50%      1
    66%      1
    75%      2
    80%      2
    90%      3
    95%      4
    98%      6
    99%      8
   100%     15 (longest request)
○ vagrant@csc3150:~/csc3150/Assignment_2_120090874/3150-p2-bonus-main/thread_poll$ 
```

10 threads, 50000 requests



```
● vagrant@csc3150:~/csc3150/Assignment_2_120090874/3150-p2-bonus-main/thread_poll$ ab -n 50000 -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 5000 requests
  Completed 10000 requests
  Completed 15000 requests
  Completed 20000 requests
  Completed 25000 requests
  Completed 30000 requests
  Completed 35000 requests
  Completed 40000 requests
  Completed 45000 requests
  Completed 50000 requests
  Finished 50000 requests


  Server Software:
  Server Hostname:        localhost
  Server Port:            8000

  Document Path:          /
  Document Length:        4626 bytes

  Concurrency Level:      10
  Time taken for tests:   5.643 seconds
  Complete requests:      50000
  Failed requests:        0
  Total transferred:      234600000 bytes
  HTML transferred:       231300000 bytes
  Requests per second:    8861.07 [#/sec] (mean)
  Time per request:       1.129 [ms] (mean)
  Time per request:       0.113 [ms] (mean, across all concurrent requests)
  Transfer rate:          40601.72 [Kbytes/sec] received

  Connection Times (ms)
                min  mean[+/-sd] median   max
  Connect:        0    0   0.4      0      32
  Processing:     0    1   1.5      1      84
  Waiting:        0    1   1.4      0      84
  Total:          0    1   1.5      1      84

  Percentage of the requests served within a certain time (ms)
    50%      1
    66%      1
    75%      1
    80%      1
    90%      2
    95%      2
    98%      3
    99%      3
   100%     84 (longest request)
○ vagrant@csc3150:~/csc3150/Assignment_2_120090874/3150-p2-bonus-main/thread_poll$
```

--- End of Report ---