Operating System:
Multi-Thread Programming Report

10/22/2022

CONTENTS

1. Development environment

The development environment for the main program (frog crosses river) and the bonus program (thread pool) are:

- Linux Distribution Version:

```
xl@ubuntu:~$ cat /etc/os-release
NAME="Ubuntu"
VERSION="18.04.6 LTS (Bionic Beaver)"
ID=ubuntu
```

Ubuntu 18.04.6 LTS

- Linux Kernel Version:

```
xl@ubuntu:~$ uname -r
5.10.145
```

5.10.145

- GCC Version:

```
xl@ubuntu:~$ gcc --version
gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
Copyright (C) 2017 Free Software Founda
This is free software; see the source f
warranty; not even for MERCHANTABILITY
```

7.5.0

2. Program Implementation

2.1 Frog Crosses River

2.1.1 Program Design

In this task, the program uses multi-threads programming to run a game in the terminal about a Frog to cross the river by jumping through the moving logs:

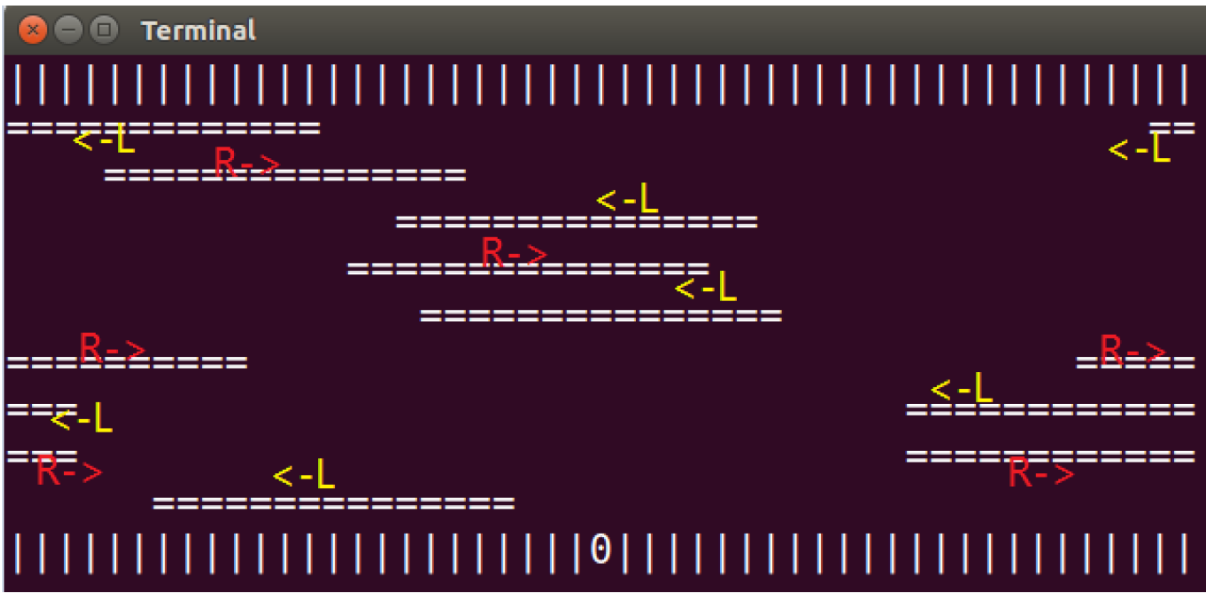

Figure 1: Game Demo

Before any implementation, the important data structures used for this program are given below.

- **Game console:** two-dimensional static array

```
char map[ROW + 10][COLUMN];
```

Figure 2: map

- **Objects:**
  - Log: ==========
  - Frog: 0
  - River bank: ||||||||||||||||||||

Figure 3: Objects

To implement, **3 threads** are created to take different functions:

```
i = pthread_create(&capture_t, NULL, user_move, NULL);
if (i != 0) {
    printf("create user_move thread failed, %d\n", i);
    exit(1);
}
j = pthread_create(&move_t, NULL, logs_move, NULL);
if (j != 0) {
    printf("create logs_move thread failed, %d\n", i);
    exit(1);
}
k = pthread_create(&display_t, NULL, display_console, NULL);
if (k != 0) {
    printf("create display_console thread failed, %d\n", i);
    exit(1);
}
```

Figure 4: Threads Creation

- **user_move** is a function that detects the user's input and identify if there are any hot keys for movement. After that, the function will update the frog's (Node) coordinates accordingly.
- **logs_move** is a function that moves the logs at certain pace (i,e, updates the map) and then moves frog if needed and check the game status.
- **display_console** is a function that simply prints out the game console (print the map) if the game is on, or prints out the relevant information according to the flag.

To guarantee the three threads are functioning properly, the program utilizes **2 mutex locks**:

```
pthread_mutex_t print_lock;
pthread_mutex_t frog_lock;
```

Figure 5: Mutex Locks

- **print_lock** is used to ensure the intactness of the map from the frequent read and write. It is used in the display_console() function to prevent any changes while the function is printing out the whole map; On the other hand, it is also used in the

logs_move() to preventing any print request while the function is updating the whole map.

- **frog_lock** is used to ensure the correctness of the frog's position. It's used in user_move() to prevent the effect of the speed and update the frog's position with respect to the user input direction; It's also used in the logs_move() function to prevent any user input while the function is updating the frog's position if the frog lands on the log and gain the same speed of log;

To check the game condition, **6 flags** are initialized for different situations:

```
int GAME_STATUS = 1;
int MOVE = 0;
int bound_flag = 0;
int water_flag = 0;
int win_flag = 0;
int quit_flag = 0;
```

Figure 6: Game flags

- **GAME_STATUS** is the flag that determines whether the game is still going. It's used as an indictor in the while loops in all 3 thread functions mentioned above. It's set as 0 when the game is over.
- **MOVE** is a flag that detects the whether the user have hit the keyboard for the first time (if not, then the program can skip some checks and run an empty loop to save some resources); it also includes the information for the moving direction. For example, moving backwards 's' sets the MOVE flag as 2.
- **bound_flag** is a flag that checks whether the frog have hit the river boundary. It's set as 1 when the condition is satisfied.
- **water_flag** is a flag that checks whether the frog have jumped into the water. It's set as 1 when the condition is satisfied.
- **win_flag** is a flag that checks whether the frog have reached the opposite river bank. It's set as 1 when the condition is

satisfied.

- **quit_flag** is a flag that checks whether the user have entered ‘q’ to quit the game. It’s set as 1 when the condition is satisfied.

Finally, as a good convention and also to re-display to cursor, the programs uses pthread_join() to wait for each thread to terminate, and then exit the whole program.

```
pthread_join(capture_t, NULL);
pthread_join(move_t, NULL);
pthread_join(display_t, NULL);
```

Figure 7: Join each thread

2.1.2 Program Execution

To launch the game, following steps are provided.

1) Open the terminal in the folder that contains hw2.cpp
2) Enter **g++ hw2.cpp -lpthread**
3) Enter **./a.out**

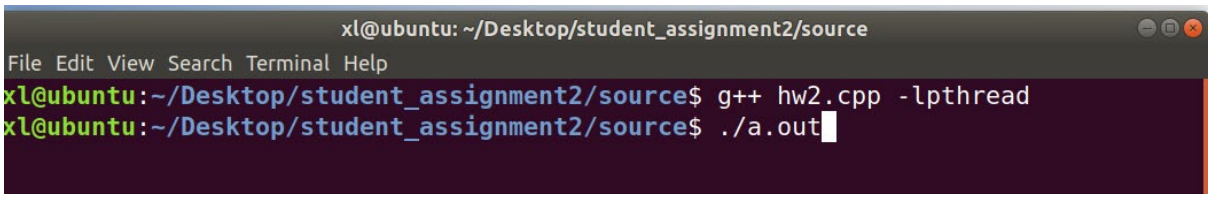

Figure 8: Program Execution

2.1.3 Program Output

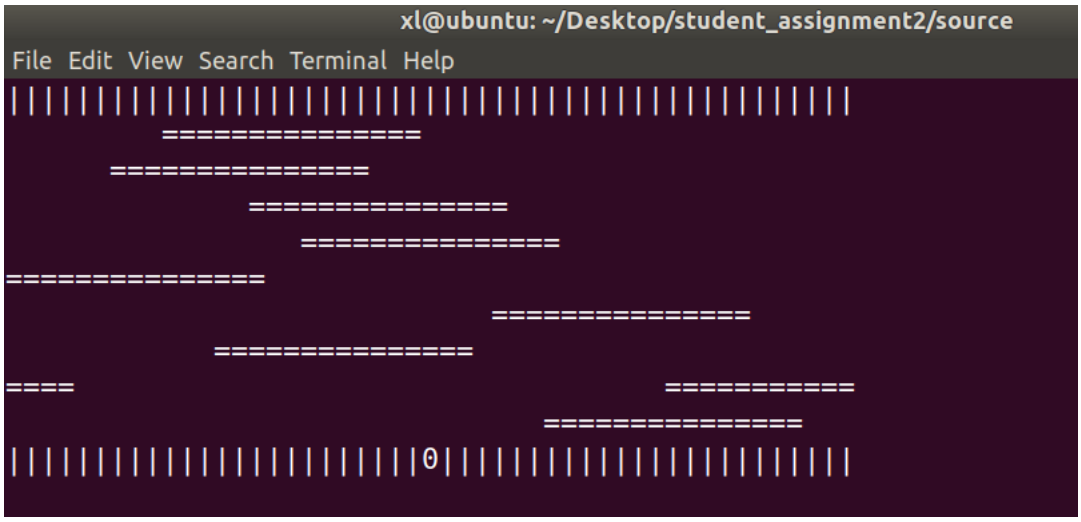

Figure 9: Start Game

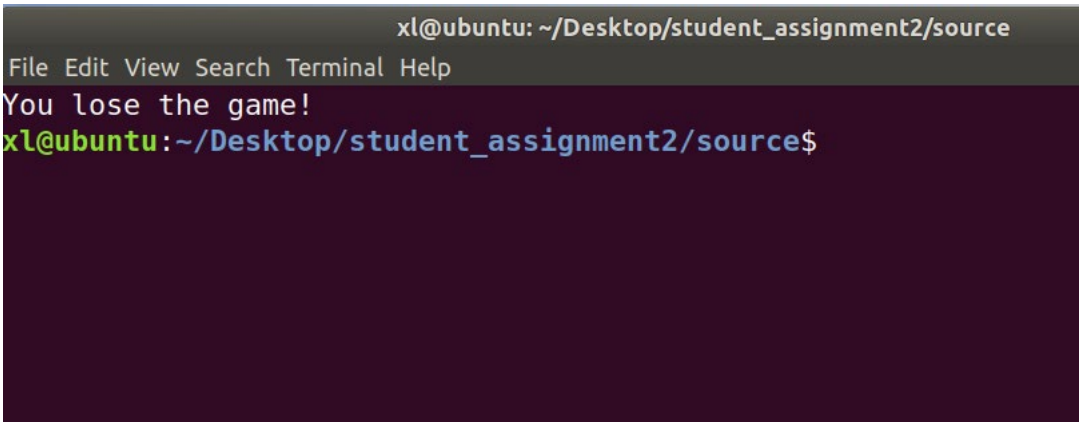

Figure 10: Lose Game

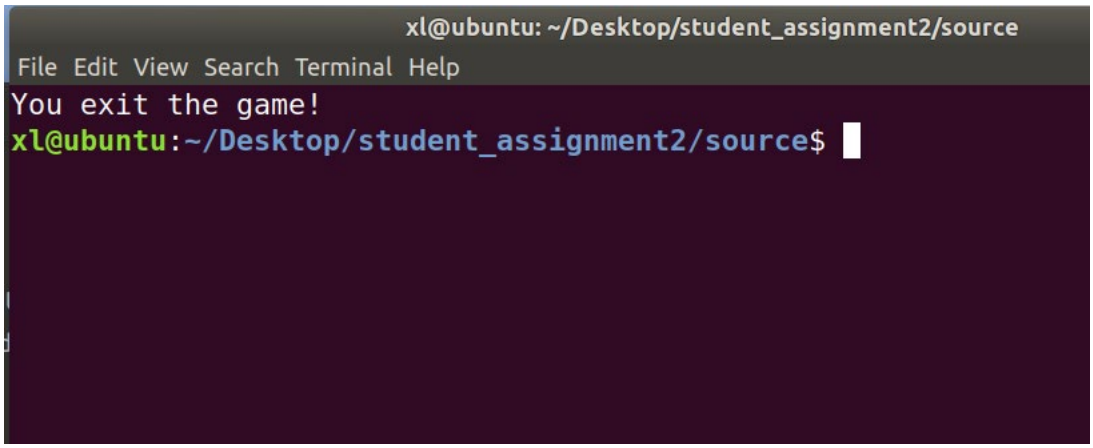

Figure 11: Quit the game

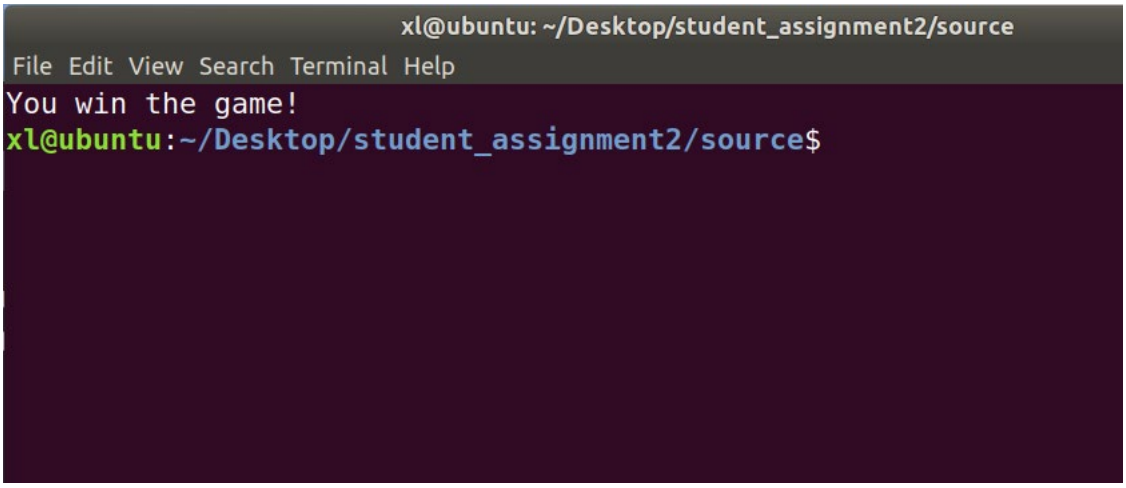

Figure 12: Win the game

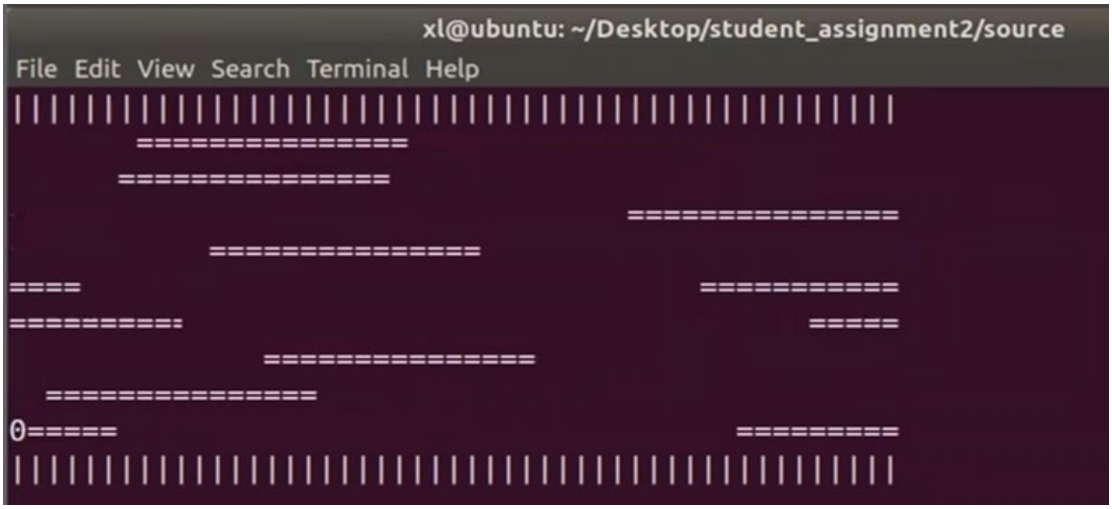

Figure 13: touch border (game over)

Figure 14: Jump into water (Game over)

2.2 Thread Pool (Bonus)

2.2.1 Program Design

This task requires to implement a thread pool that can handle the target job only if there is any; otherwise, the threads need to go sleep, and no busy waiting is allowed.

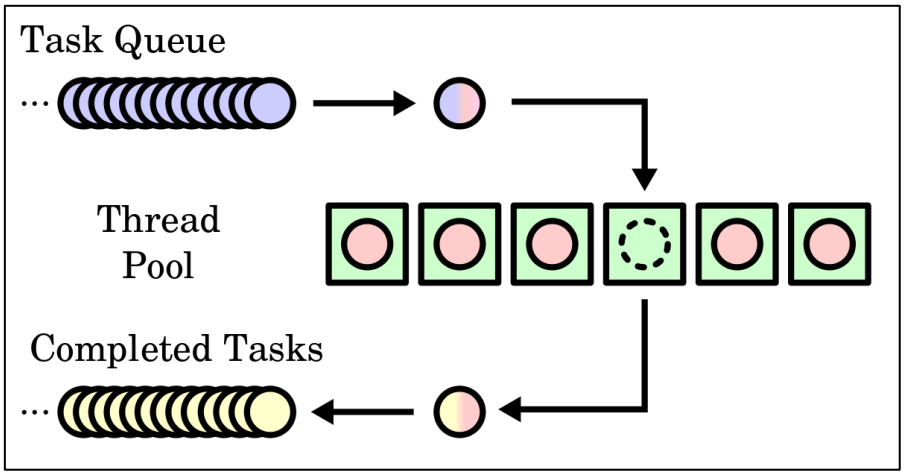

Figure 15: thread pool

To implement, the important **data structures** of the thread pool are given below:

- Task: nodes inside a doubly linked list.

```
typedef struct Task {
  struct Task *next;
  struct Task *prev;
  void (*target_function) (int);
  int arg;
} Task;
```

Figure 16: Task structure

- task_queue: a doubly linked list with head and size.

```
typedef struct task_queue {
  int size;
  Task *head;
} task_queue;
```

Figure 17: task queue

The design logic is quite simple. First, the program use async_init() to create certain number of threads, and assign each of the thread to the starter() function.

```
for (i = 0; i < num_threads; i++) {
    res = pthread_create(&pool[i], NULL, &starter, NULL);
    if (res != 0) {
        perror("Failure to create the thread");
    }
}
return;
```

Figure 18: create threads

Then the thread with starter function will check if there's anything in the task_queue. If no, the thread will go to sleep, release the mutex lock and wait for the signal.

```
while (1) {
    Task *this_task;
    pthread_mutex_lock(&queue_lock);
    while (my_queue->size == 0) {
        pthread_cond_wait(&wakeup_call, &queue_lock);
    }

    this_task = my_queue->head;
```

Figure 19: conditional variable

When calls async_run(), the program will create a Task with a pointer to the target function and its arguments, and then append this task into the task queue and returns.

```
Task *ins_task = (Task *)malloc(sizeof(Task));
assert(ins_task);
ins_task->target_function = handler;
ins_task->arg = args;
ins_task->prev = NULL;
ins_task->next = NULL;

pthread_mutex_lock(&queue_lock);

DL_APPEND(my_queue->head, ins_task);

my_queue->size += 1;

pthread_mutex_unlock(&queue_lock);
pthread_cond_signal(&wakeup_call);

return;
```

Figure 20: create & append task

2.2.2 Program Execution

1) Install “ab - Apache HTTP server benchmarking tool” in Linux via **sudo apt-get install -y apache2-utils**
2) Open the terminal in the bonus folder contains async.c, async.h and other provided files.
3) Enter **make**
4) Enter **./httpserver --files files/ --port 8000 --num-threads T** (replace T with a thread number, say 10).
5) Open another terminal and enter **ab -n X -c T http://localhost:8000/** (replace X with the total request number, say 5000, and T with the thread number, say 10)

2.2.3 Program output

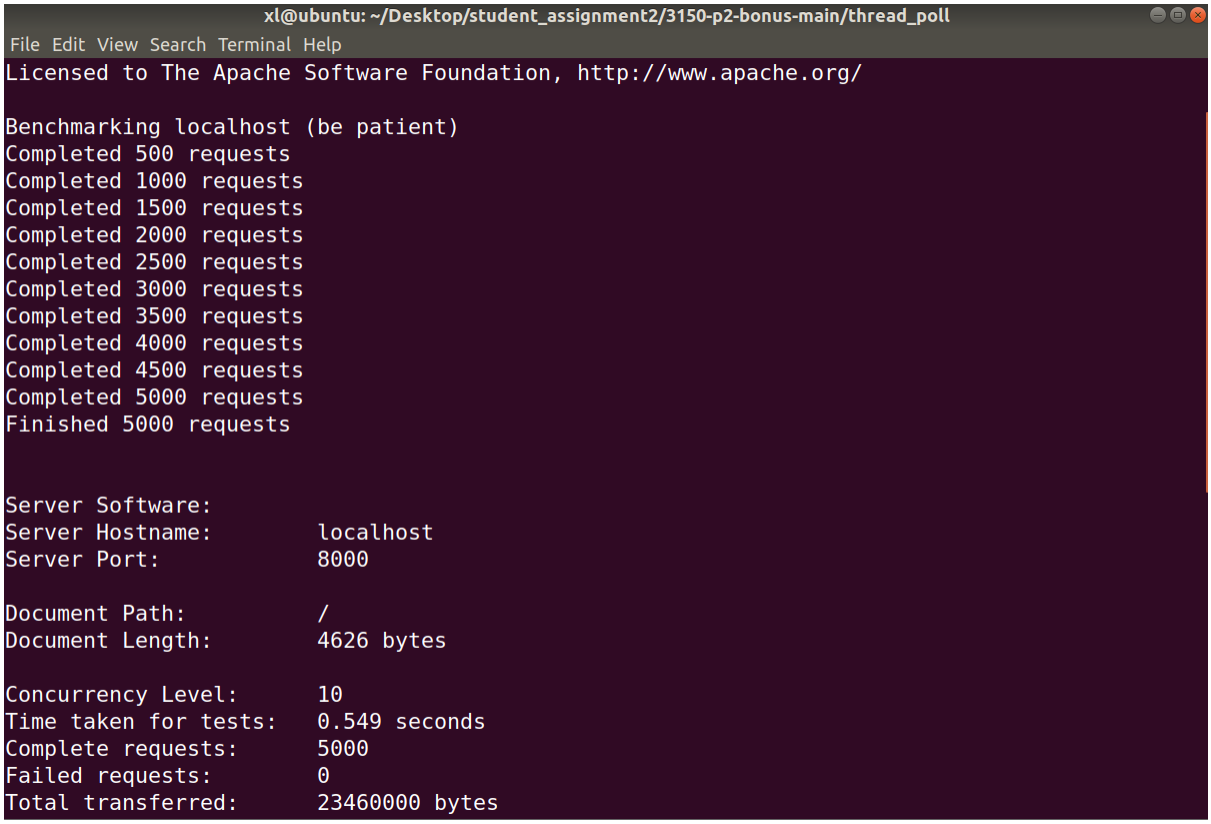

Figure 21: bonus output 1

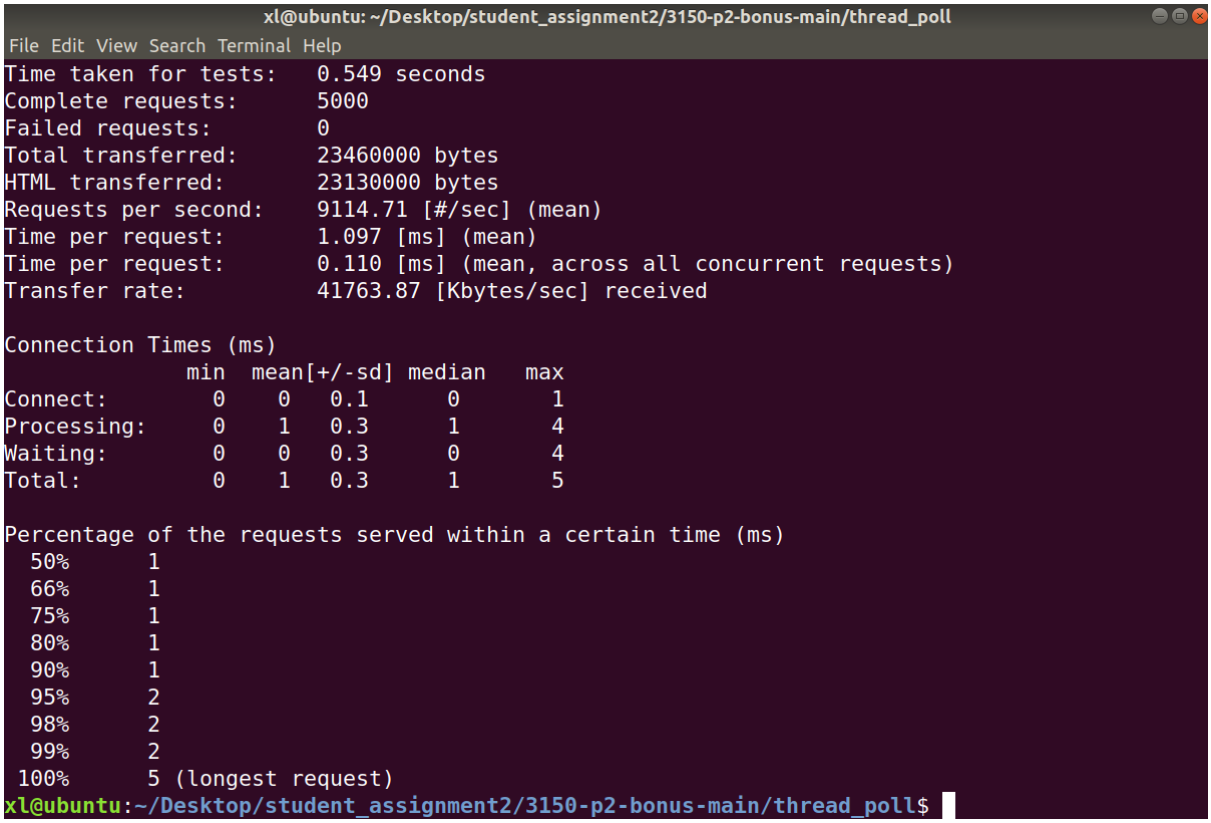

Figure 22: bonus output 2

3. Learning outcome

Through this program, **I've learned**:

1. How to employ appropriate functions and tools, such as join function, mutex, and conditional variables, in the multi-thread programming.
2. Understand the behind mechanism for the mutex, conditional variables, and atomic operations.
3. How to adjust game parameters for a better user experience.
4. How to initiate a thread pool with asynchronous operations that can greatly lower the CPU usage.
5. How to debug for multi-thread program using GDB.