CSC3150 Assignment 2

Chen Zhixin, 120090222

CSC3150 Assignment 2
- How I design my program?
  - Main
    - Too long don't read
    - Keep atomicity
    - Critical situation
  - Bonus
    - Too long don't read
    - Waiting list
    - Avoid keep spinning
- Environment
- The steps to execute my program
  - Main
  - Bonus
- Screenshot of the program Output
  - Main
  - Bonus
- What did I learned from task

How I design my program?

Main

Too long don't read

There are totally 11 threads in the main program. They loops forever until the frog reached the opposite river bank.

- 9 x `logs_move` Thread
- 1 x `Print_Map` Thread
- 1 x `Controller` Thread

At first, the `controller` thread will go to sleep. Now the loops remains 10 valid threads (9 x `logs_move`, 1 x `Print_Map`).

For each `logs_move` thread, it will analyze one bit of keyboard input:

- If `Q` detected, wake the `controller` up and quit the game.
- If `W` detected: Moves the frog **upward** (If the frog step into the water, quit the game)
- If `A` detected: Moves the frog **leftward** (If the frog step into the water, quit the game)
- If `S` detected: Moves the frog **downward** (If the frog step into the water, quit the game)
- If `D` detected: Moves the frog **rightward** (If the frog step into the water, quit the game)
- If the frog reached the opposite river bank, wake `Controller` Thread up.

Then the corresponding # log moves leftward/rightward for one unit length.

And the `Print_Map` will clean the screen, display the new map.

Once if the `Controller` Thread woke up, It will ask all other threads to quit (suicide).

Keep atomicity

Compare to the traditionally c++ codes, the difference is far from simply adding -lpthread label when compile - It involves parallel programming, that is, using multiple threads to compute and get result. For each thread, their task can be simplify as below:

- Fetch data from shared memory
- Execute
- Put(store) data back to shared memory

A vivid example is the bank transaction system. Actually it's an large-scale distributed parallel system. Thousands of people are doing transactions every where. For example, if someone want to fetch money from his(or her) bank account, then first it need to check the balance. If he/she has enough money to pay, then the bank account will be deduced, and he/she can successfully get the money. But what if he/she tries to fetch money from two different ATMs at the same time? Think about that, first step is to check the balance. If he/she only has $100 balance, and try to take out all what he/she has from two machines. Since they check the balance at the same time, if there are no protection on the data, then two machines think he/she has enough money. Then, he/she just get $200 from it. Seems amazing, isn't it? He/she got twice of what he has, thanks to the bugs of the transaction system. How to avoid this from happening again? The answer is to add the exclusive lock when doing sensitive modification on shared memory. For the fetching money case, the shared memory is the bank account. The correct thing is that, when one ATM is requiring the balance, the other should wait until the requiring and fetching process done, then check the balance. That is equivalent to add some "lock" on it: once if the lock added, it is protected, no one else can check or modify the shared memory, until the lock holder itself unlock that, then the next one waiting for that lock carry on.

As for our program, these two variables are regarded as shared content, which can be seen by all threads:

- The graph/state of the river, frog, log and frog.
- The position of the frog, in the format of (x,y).

Each time the thread modify these shared content, it should be guaranteed that no one else can modify it. Otherwise, there may be severe problem, even causing dead locks. So in our program design, each thread is under the protection of mutually exclusive lock when modifying these sensitive shared content. After it's completely done, the thread release the lock and let those late-comers continue.

Critical situation

Except for the `controller` thread, which falls asleep at the first loop, the remaining 10 threads will keep running in an endless loop. Every time one of the `logs_move` thread is executed, the corresponding log will move rightward or leftward for one unit length. So here comes several problems:

- How can the frog moves?
- How to know the frog fails or success?
- How can the log moves?
- How to ask all the threads quit the endless loop
- How can we control the speed of the log?

For the first problem, as mentioned above, the position of the frog is stored in the shared content, in the format of (x, y), which reflects the location in the game map. Each time when the `logs_move` thread runs, it will read one bit of letter from the input stream, then it directly modify the position (x, y) in the shared content, so that later on when it's printed out, the frog's position correspondingly changes.

The second question, all objects, including the river bank, logs, frog and boundary, there position are also stored in the shared content, but haven't printed out. So the easiest way is, when the frog moves, see if the next block frog is moving in is the river. If yes, then quit the loop and end this game. Otherwise, let it be.

As for the third question, a log is set to be 15 units length by default. Since when the frog standing on the log, we need to make the frog move as the log moves, therefore we need to move logs leftward/rightward bit by bit. Just use a for loop, from the starting point of the logs(which is stored in advance) move the log by one bit. If the log cross the leftmost/rightmost of the game boundary, then use `%` to get the new location of the log segment.

The game status depends on the location of the frog. If the frog block into the water, or, the frog block out of the boundary(no matter follow the floating logs or do it by itself), the game will fill. And when the user press q, the game will be quit. When the frog reaches the opposite bank of the river, the game success. When the game is interrupted, no matter what kind of reason, the sleeping `controller` will be wake up, and the `stop_signal` will be set. Once the `controller` is wake up, the `stop_signal` is set, since each thread need `stop_signal` to achieve endless loop, one set, each thread will end their loop, eventually exits the route. Otherwise, the quit status is not achieved, the logs will move forever.

Last, after each thread's execution in a while loop, the process will wait for a short while with `usleep(sec)`. By controlling how long they wait, we can decide how fast the log moves.

Bonus

Too long don't read

I implemented thread pool by **producer-consumer pattern**.

- There is a waiting list, all the pending tasks are stored in it.
- Each thread is the **consumer**. They execute the remaining tasks in the waiting list one after another. If **consumer** has no task to execute, it will go to sleep.
- `async_run` is **producer**. It will pass the task into the waiting list. After **producer** pass in a new task, it will randomly wake one **consumer** up.

Waiting list

I didn't used the data structure provided. Instead, I designed it on my own. I implemented an queue structure, which is easy for me to insert at the head and delete from the tail. For each consumer, it will produces one task. At that time, the task will be packed as the structure 'item' and throw it into the tail of the queue. Then the `async_run()` will return immediately, making it a asynchronized function (Notice that it is called asynchronized because the `async_run()` only need to throw the task into the waiting list, but not to directly calculate it). Then, the hungry consumers will fetch these task from waiting list, and delete them from the waiting list..

Avoid keep spinning

At the first time I implement it, I made a mistake. I let those hungry consumers keep searching if there is anything in the waiting list. The potential problem is, when no request coming in, these threads still spinning over and over again, which brings a log of burden for the system resources (It takes up ~100% of my CPU resources!). The ideal case is, when there is no task coming in, the consumer threads should fall asleep. Next time a new task is added to the waiting list (i.e. the `async_run` is called), the consumer threads should wake up and handle them. So I made some important changes: to add the conditional variable to wake those threads up. For each thread, if it sees no tasks in the waiting list, it will wait for signals in conditional variable, meanwhile to release the mutually exclusive lock. Then after several CPU cycles, all the threads will fall asleep. Next time the tasks crowded in, the `async_run` call the `pthread_cond_signal` to randomly pick one thread to wake up, the lucky strike being selected will re-enter the endless loop. With more and more tasks flourish in, the threads are wake up one after another, until all of them get to work. This mechanism can greatly reduce the cost of the system resources when no tasks for a long time, and Increase system persistence.

Environment

OS Version: Ubuntu 20.04

Kernel Version: 5.10.x

Check g++ version > 4.9

```
g++ --version
```

Check kernel version ~5.10.x

```
uname -r
```

(Optional) Install `libncurses5-dev`

```
sudo apt-get install libncurses5-dev
```

The steps to execute my program

Main

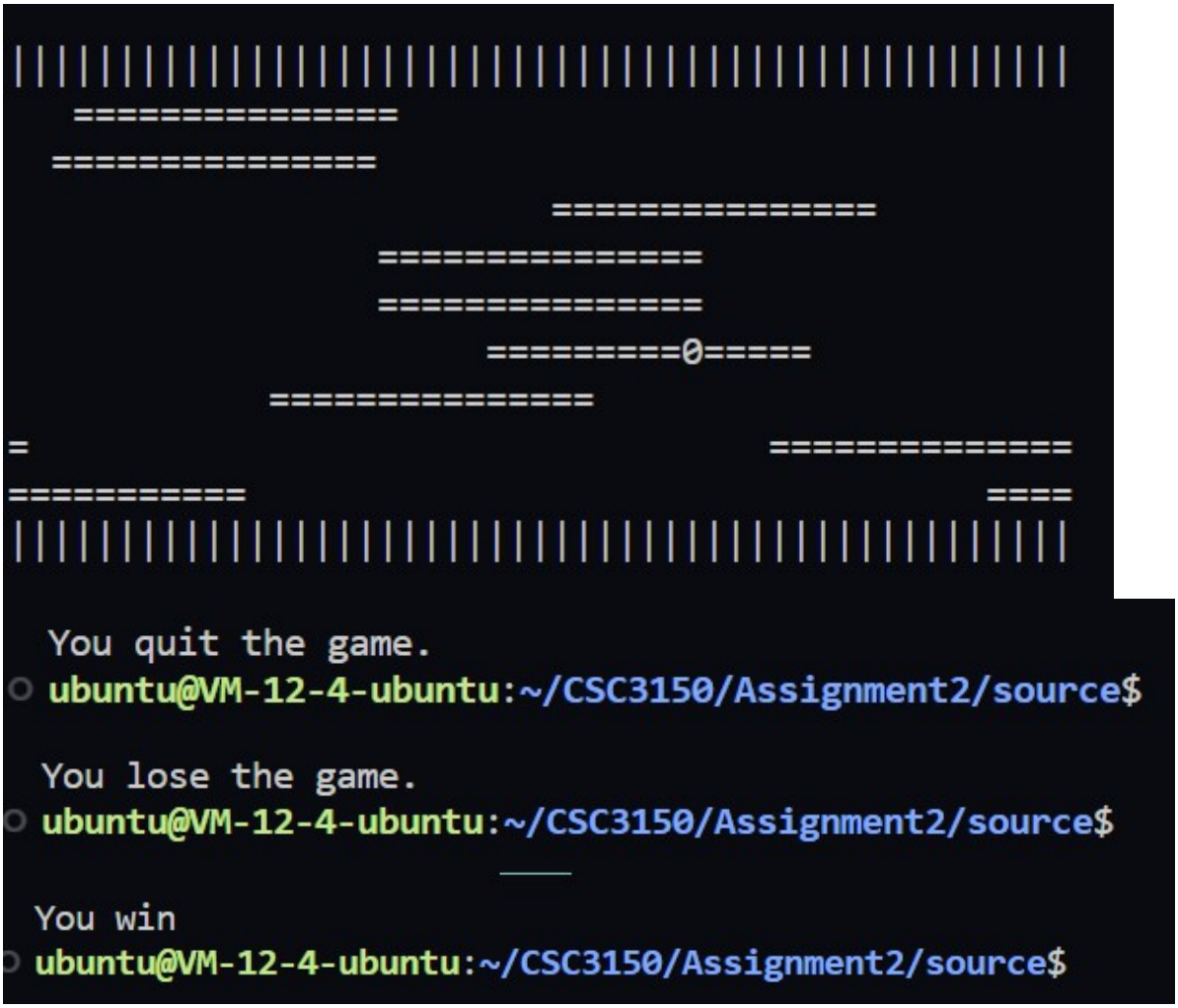
```
cd ~/source
make
./a.out
```

Bonus

```
cd ~/3150-p2-bonus-main/thread_poll/
make
./httpserver --files files/ --port 8000 --num-threads 10
ab -n 5000 -c 10 http://localhost:8000/
```

Screenshot of the program Output

Main



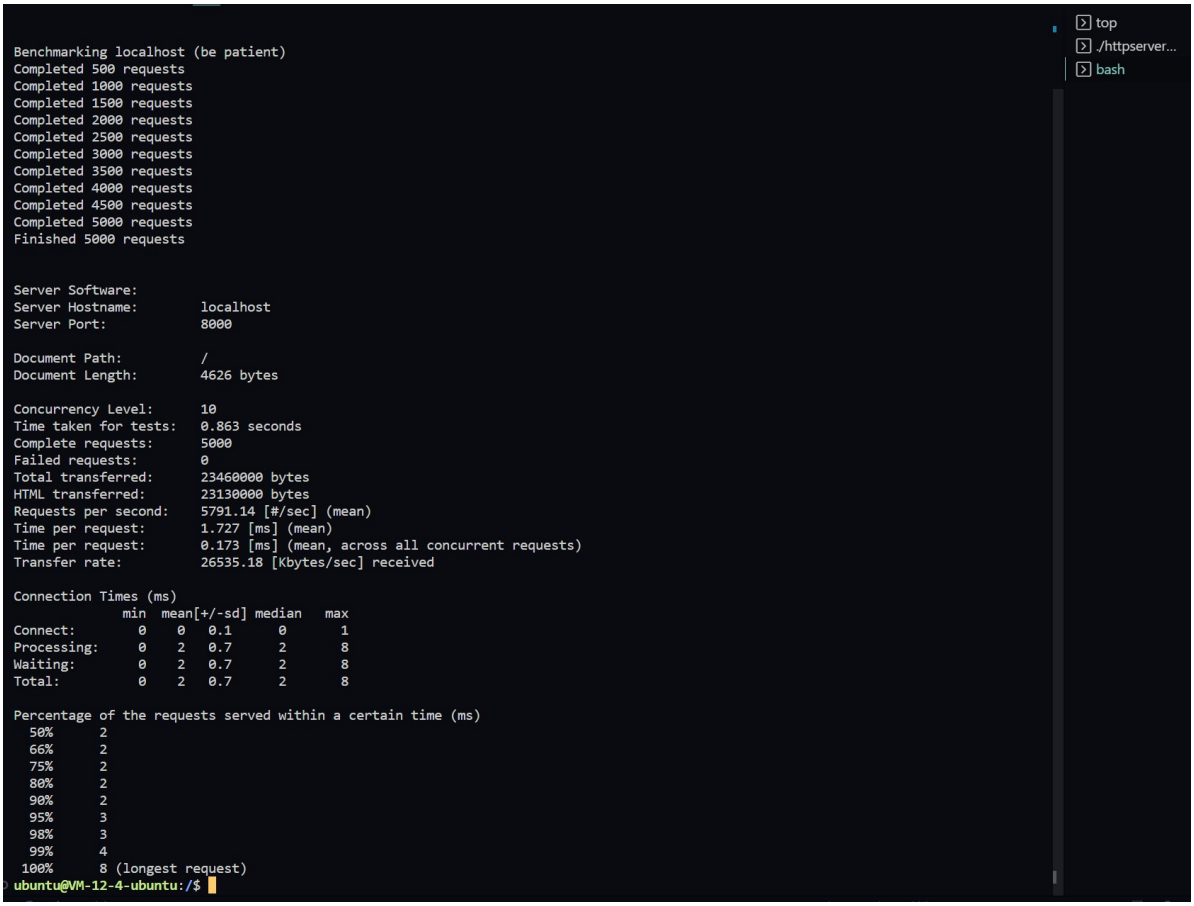
If you wanna see the ***Video for DEMO***, click **HERE**

> Video may get stuck due to video server bandwidth limitations.

Bonus

With AB benchmark test

10 Threads, 5000 requests:



What did I learned from task

Concurrency is a very important topic in OS. In this task, I implemented a lot of features relative to mutual exclusive lock, conditional variable and so on. Hence I deeply understand how important it is to handle the concurrency conflict in parallel calculation among multiple cores. Also, during a lot of debugging work, I learned new techniques of figuring out where exactly the bug is by some automatic tools like `gdb` or `objdump`. Also, I can analyze the program flow with state map, which helps me to understand the process, and find out the logical fault of the program.

Some strategies I used:

- Conditional Variable
- Mutual exclusive lock
- Keep **Atomicity**, protect shared memory
- Draw State Graph to avoid dead lock
- Use automatic debug tools(`gdb` , `objdump` etc.)