CSC 3150 Assignment 2 Report

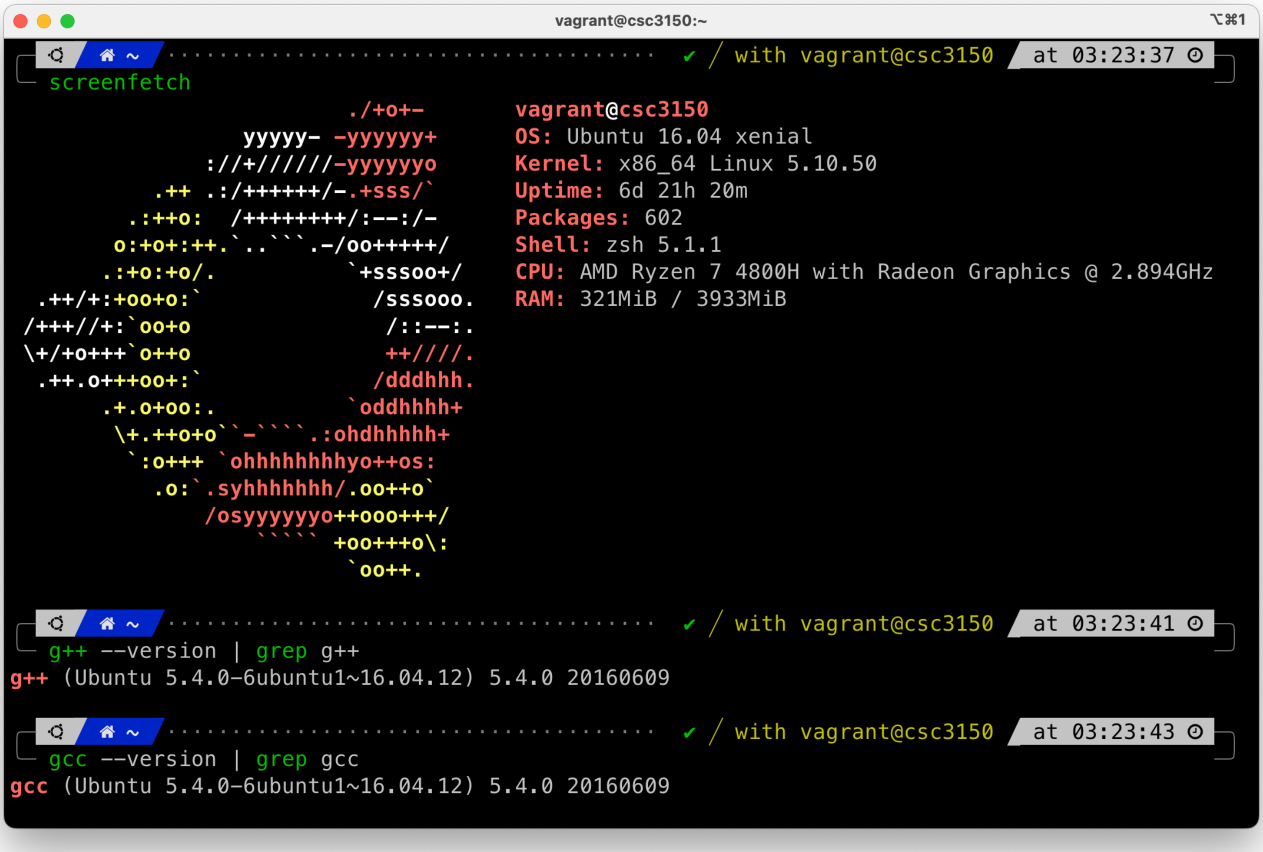
120090472 Tianhao SHI

Oct, 2022

Enviromnent and Execution

Environment Info

This assignment is written, compiled, tested in the following environment



Steps to execute

For the frog game:

```
$cd ~/Assignment_2_120090472/source    #navigate to the code directory
$make                                  # compile and run

# (optional)
#$ make build                          # compile only
#$./hw2                                #run
```

For the thread pool

```
$cd ~/Assignment_2_120090472/3150-p2-bonus-main/thread_poll # I don't know if there's a typo
$make          #compile
$./httpserver #run
```

Task 1

Core Data Structures

The `frog` struct

I modified the provided `frog` struct. Its members `x` and `y` are changed to `row` and `col` for consistency conserns.

The `logs_left_arr`

This global array of integers with size 9 stores the column information of each log's left edge. The right edge's location can be acquired using the `log_right` macro.

The board

The game board is a 2d array with size `ROW+1` x `COL`. This corresponds with 2 banks and 9 logs

Arrangements of threads

This program will create two threads other than the main thread, the `logs_thread` and the `frog_thread`. The former thread controls the movements of logs. It will also let the frog follow the log's movement when the frog jumps on a particular log. The latter thread is used to listen to keyboard inputs that allows players to control the frog. The `frog_thrad` thread is also responsible for updating the game status. Both threads will update the game board whenever the position of frog or log is updated.

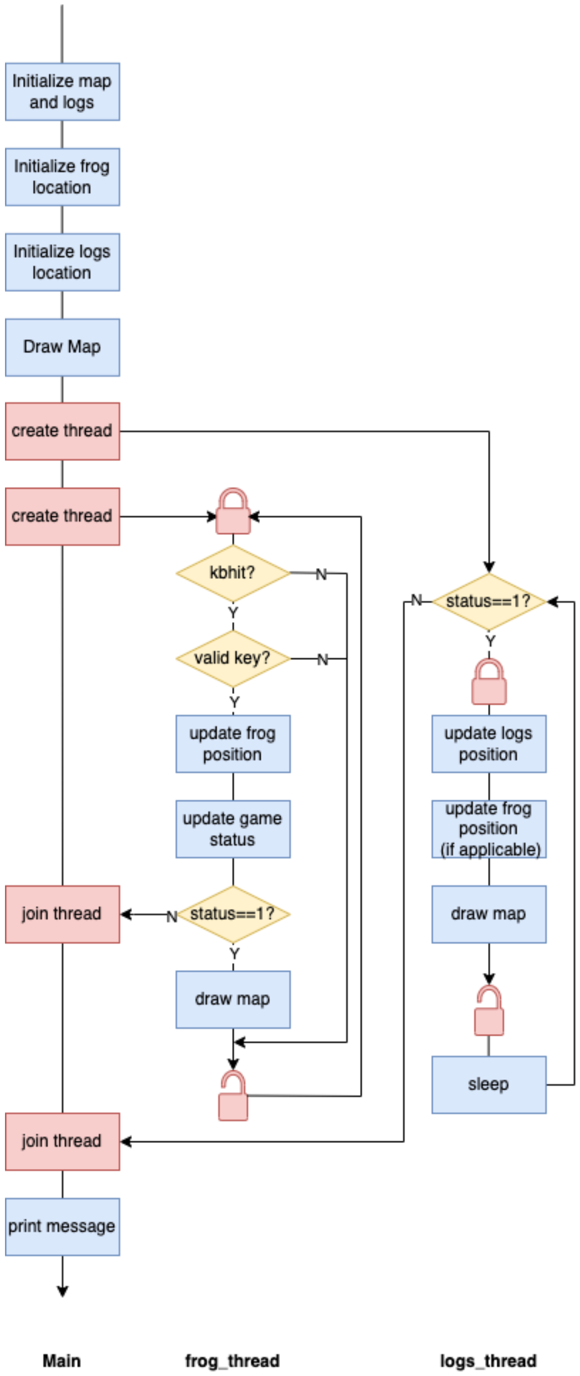
Both threads will be quitted based on the value of global variable `status`. The value of `status` corresponds to the following:

- 1: running
- 0: game lost
- 2: game won
- 3: user quit

The main thread will first initialize the frog and log's position. Then it will first print the game board. It will then create the two threads and a mutex lock (to be discussed later in ths report). The two threads are then joined, and main thread will be blocked until these two threads are exited. It will then analyze the game status and print the corresponding message.

A mutex is created to guaranteen thread safety. For the frog thread, it will acquire this mutex lock before examining the `kbhit()` and release it after completing drawing the map. For the logs_thread, it will acquire a lock before updating the log position and after completing drawing of the map.

The following graph briefly explains the arrangement of threads in this program:



Control of the frog

the frog is controlled by the keyboard. During execution, the `frog_thread` will query if a key have been pressed by calling the provided `kbhit()` function. When this function returns 1, the pressed character will be read from the stdin using `getchar()`.

Movement of logs and frog

The logs' position will be update for a fixed time interval (defined by the `LOG_SPEED`). The elements in the `logs_left_arr` will all be increased/decreased by 1 every time they get updated. The actual increase or decrease is determined by the parit of the row index. The logs on the odd rows (that is, on the 1, 3, 5, 7, 9) will move right (increase column index), while the logs on the even rows will move left. Whenever the log's left bound hits index 0 or `ROW-1`, it will be rounded to the other side of the board. The design of the `log_right`macro also take the boundary into consideration.

When a frog is on one particular log, its position will be incremented or decremented the same as the log it rests on.

Judging the game status

In the frog_thread, after analyzing the input key and updating the frog's position, the `update_game_status()` function will be called. The game status is determined by : 1) Whether the user want to quit; 2) Whether the frog have reached the upper bank or falls into water.

Game Quit

When the user presses *q* or *Q*, the status is directly updated to 3, and the while loop in the `frog_move` is breaked. The mutex lock is released, allowing logs_thread to respond and quit.

Game Won

The game can only be won when the frog reaches anywhere on the upper bank. Therefore, by checking whether `frog.row==0` we can know if the game is won or not.

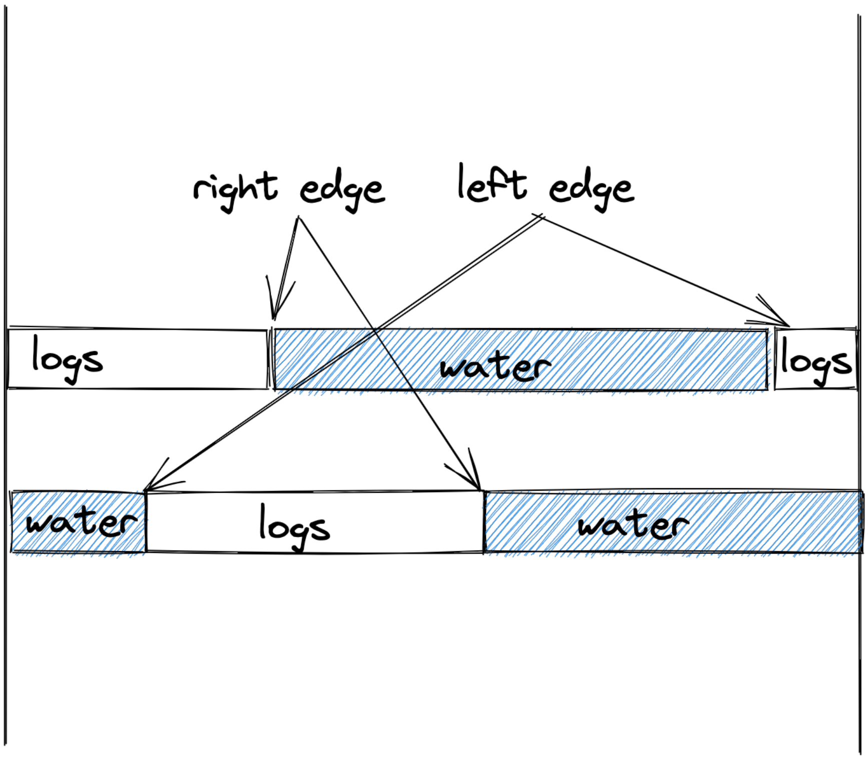
Game Lost

The game can be lost in two ways:

1. The frog hits the left or right boundary of the game
2. The frog falls into water

If the frog is in the river (0<`frog.row`<ROW) and hits the boundary (`frog.col`==0 || `frog.col`==ROW-1), then it hits the boundary.

The second cindition can be checked by examining relative position between the frog and the log on the same row. If the log doesn't wraps around the board, then when the frog's column number is smaller than the left edge of the log, or when it's greater than the right edge of the log (`frog.col` < logs_left[frog.col-1] || frog.col > log_right(logs_left[frog.col-1])), then it will fall into river. Or, if the log wraps around the board, then the frog will only fall into river when its colunn number is both smaller than the left of log **AND** greater than the right of log. The following graph illustrates my point:



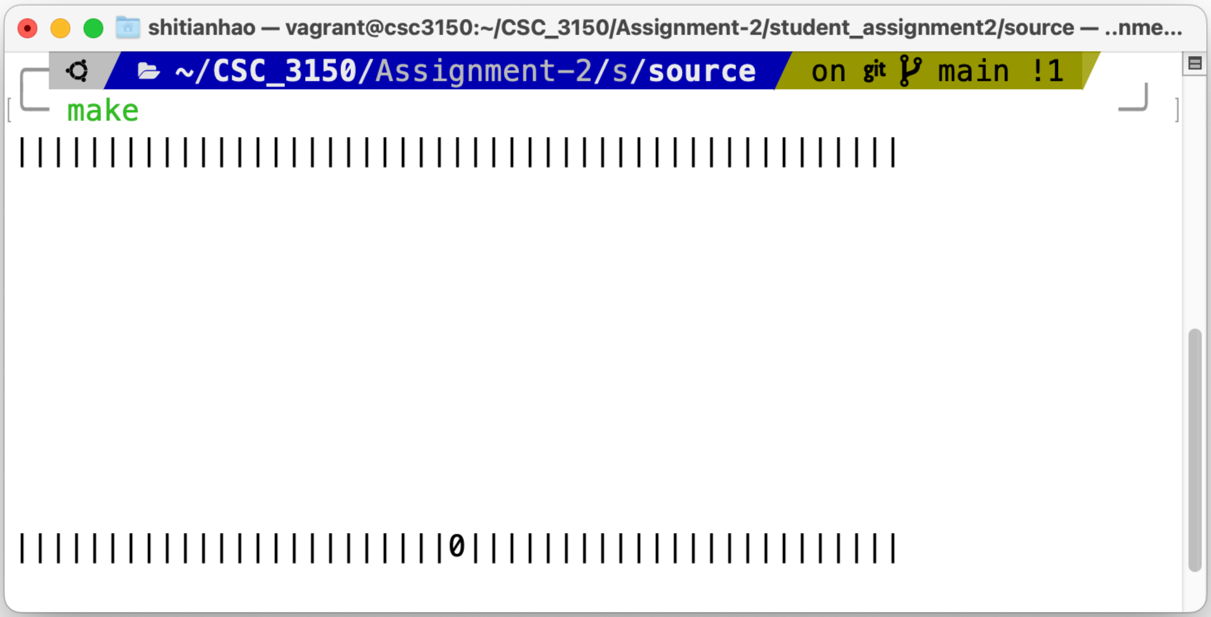
Whenever the game is won or lost, the `update_game_status()` will change the value of the global variable `status`. If `status!=1`, the `frog_thread` will release the mutex lock and break from the loop.

Print out the message

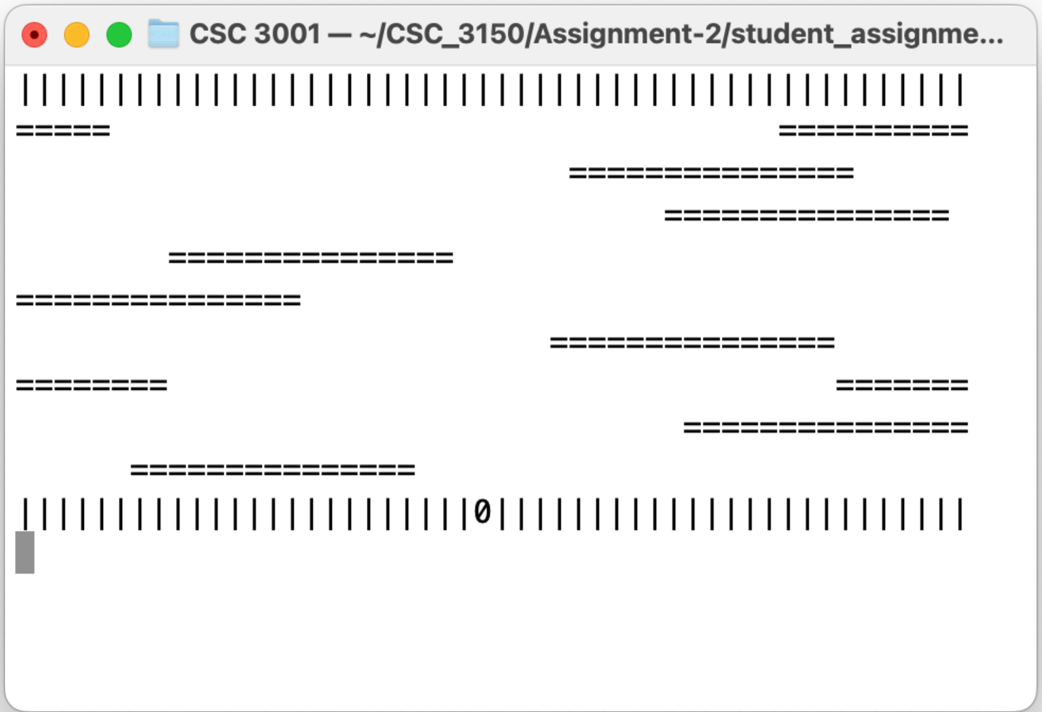
This step is fairly straightforward. After the `frog_thread` and the `logs_thread` is quitted, the main thread will print out message based on the value of the `status` variable.

Program Output

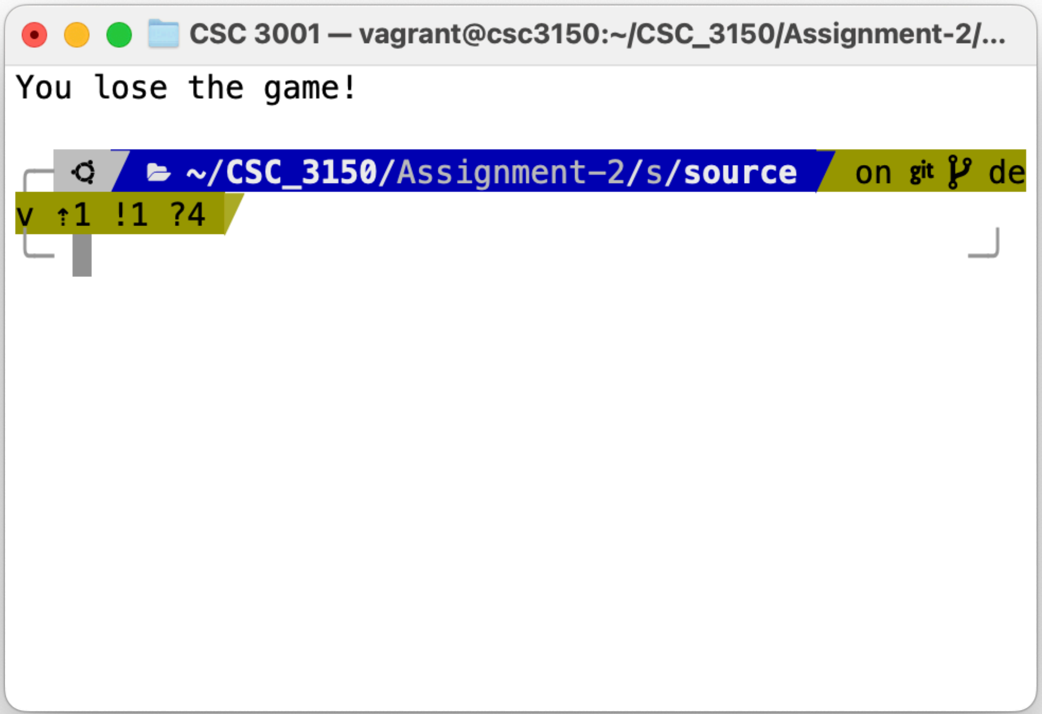
The static output of program (before any modification):



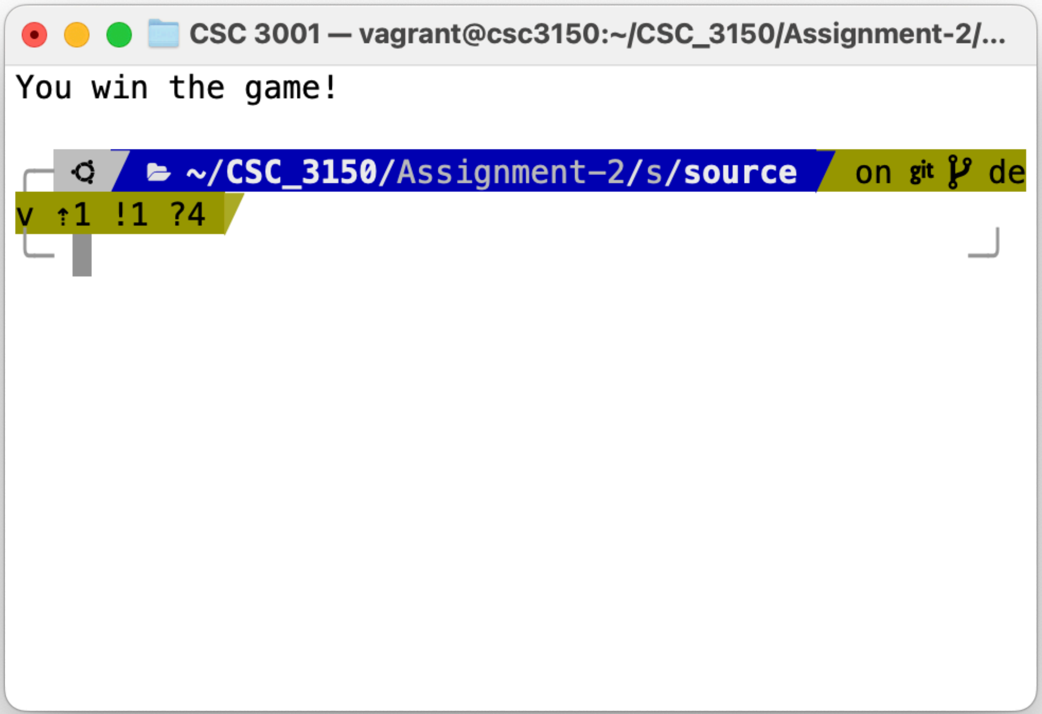
The output after starting the game



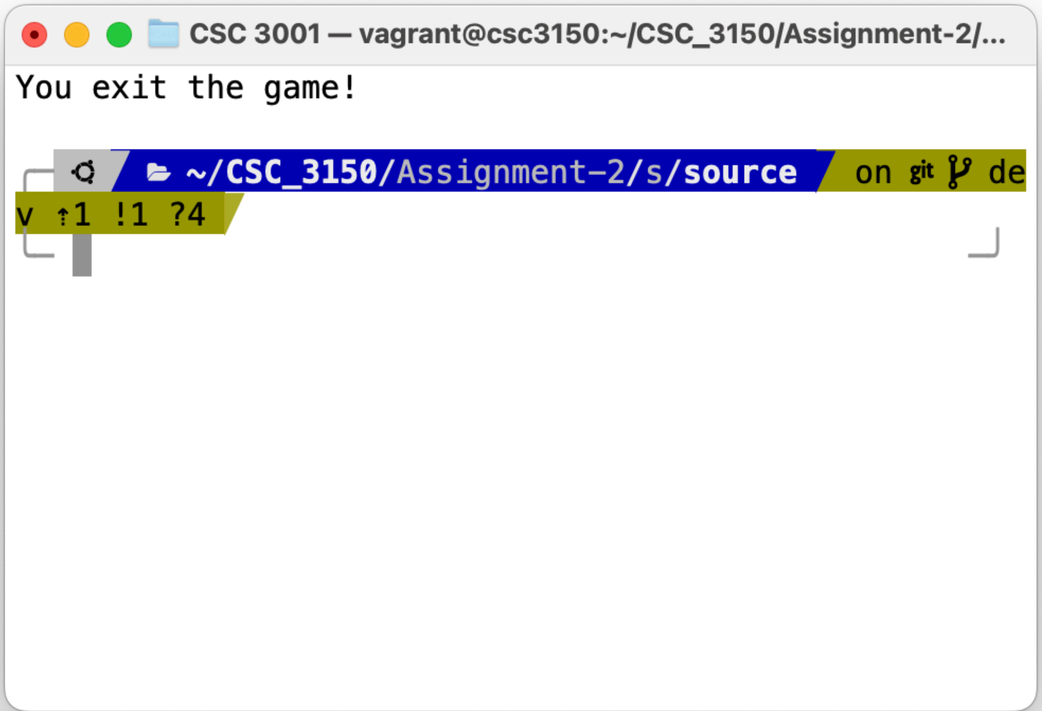
Losing the game



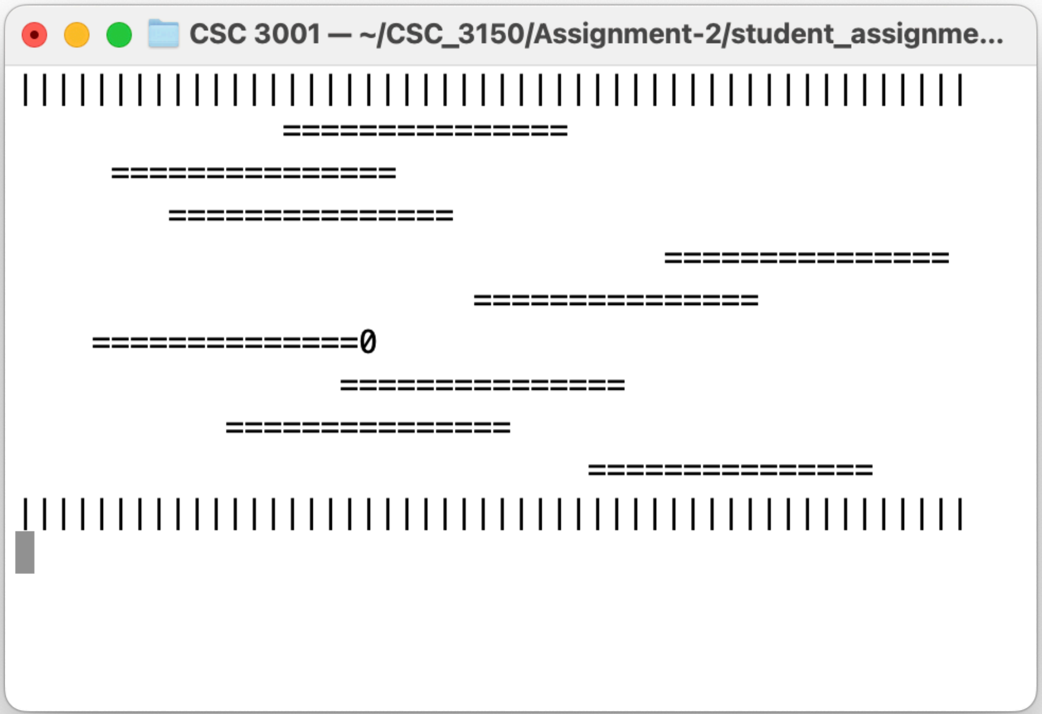
Winning the game



Quitting the game

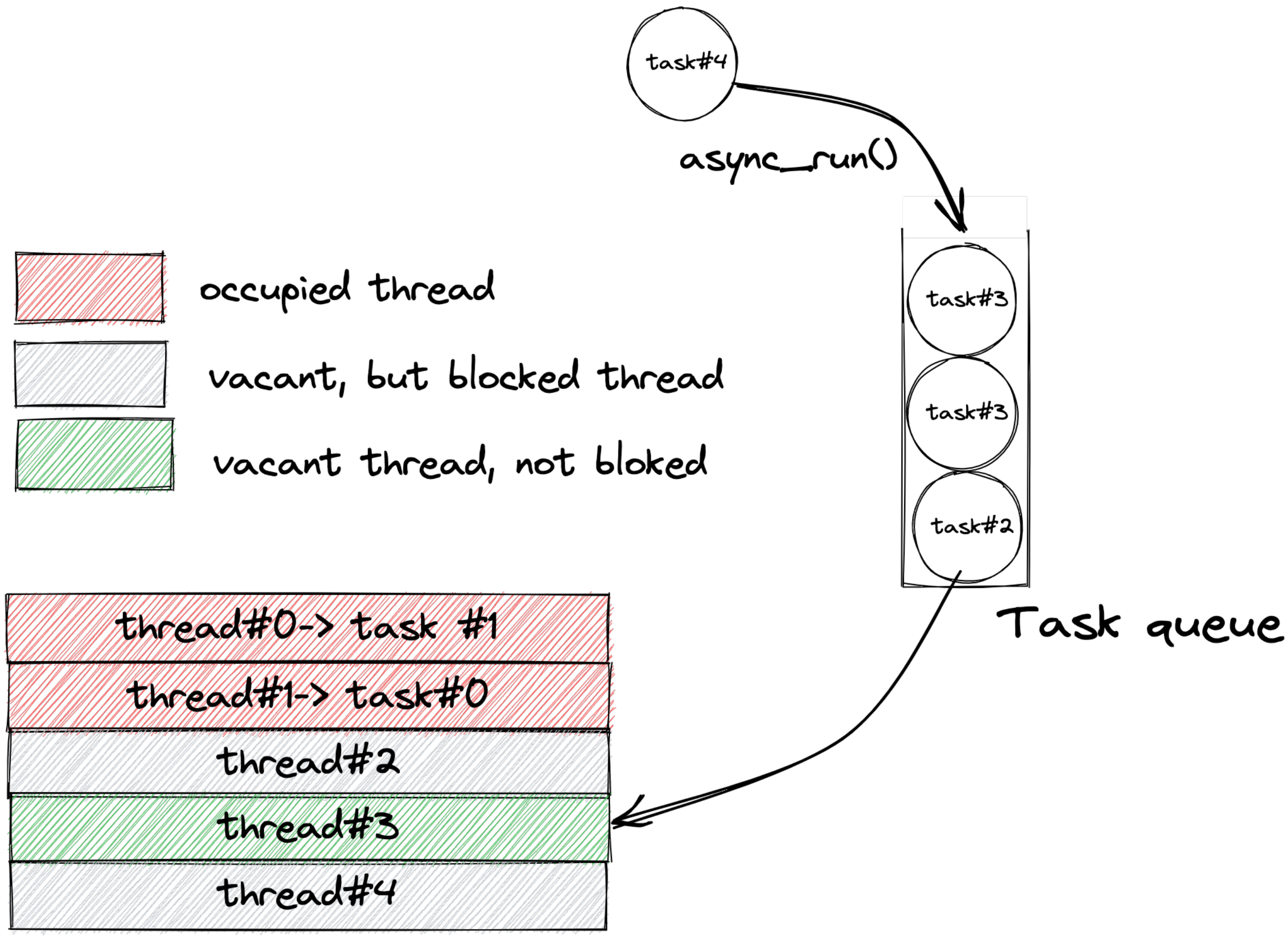


During the game



Task 2

The idea of a thread pool



The thread poo consists of two structures, the threads that run tasks concurantly, and a task queue that holds the tasks to be executed. Whenever user calls `async_run`, a new task is added to the tail of the task queue. Whenever there is a vacant thread, it blocks other potentially vacant thread and pops the head task from the queue (given the queue is not empty).

Core Data Structure

The task item

Each task item represents a task to be executed by the thread pool. It contains the function to be executed by a thread and its corresponding arguments. It also contains a `next` and `prev` pointer, which is used when it's added to the task queue

The task queue

Different tasks jointly forms the task queue. This queue will be located in the heap memory. Its size grow whenever the `async_run` function calls. The function adds a new node to the task queue. The thread pool pops item from the queue whenever there are vacant threads in the pool (and there are tasks to do in the queue)

The `work_container()` function

When threads are created, a start routine must be provided. This function will be executed during the whole life span of the thread. However, this start routine is NOT the tasks that a thread needs to execute.

Therefore, I introduced a function called `work_container`. This function does the following things:

1. Acquire a mutex lock (prevent other vacant threads from popping queue at the same time)
2. If the queue is empty, block until it recceives the signal that the queue is no longer empty (this prevents busy waiting, and will be discussed later)
3. Pop a task item from the queue
4. Release the lock
5. Execute the given task

Code design

The `async_init()` function

This function does the following things:

1. Initialize the task queue in the heap memory
2. Create the threads and detach them
   - These threads are detached because there is no point joining them, and thus they must free their resources when exited (although in this homework we didn't write methods to terminate a thread pool)

The `async_run()` function

This function is called whenever user/server receives requests and need to create a new task. Specifically speaking, this function does the following tasks:

1. Acquire a lock so that the queue is not accessed during the addition of task
2. Create a new task containing the function to be executed and its arguments
3. Append the new task to the end of queue
4. Release the lock

Program Output

PORTS 1 OUTPUT TERMINAL PROBLEMS DEBUG CONSOLE

```
Process 22099, thread 139765527598848 will handle request.
Accepted connection from 127.0.0.1 on port 1226
Accepted connection from 127.0.0.1 on port 1738
Accepted connection from 127.0.0.1 on port 2250
Process 22099, thread 139765544384256 will handle request.
Accepted connection from 127.0.0.1 on port 2762
Process 22099, thread 139765561169664 will handle request.
Accepted connection from 127.0.0.1 on port 3274
Process 22099, thread 139765552776960 will handle request.
Process 22099, thread 139765519206144 will handle request.
Process 22099, thread 139765561169664 will handle request.
Process 22099, thread 139765502420736 will handle request.
Accepted connection from 127.0.0.1 on port 3786
Process 22099, thread 139765569562368 will handle request.
Accepted connection from 127.0.0.1 on port 4298
Accepted connection from 127.0.0.1 on port 4810
Process 22099, thread 139765510813440 will handle request.
Process 22099, thread 139765535991552 will handle request.
Accepted connection from 127.0.0.1 on port 5322
Process 22099, thread 139765527598848 will handle request.
Accepted connection from 127.0.0.1 on port 5834
Accepted connection from 127.0.0.1 on port 6346
Process 22099, thread 139765552776960 will handle request.
Process 22099, thread 139765519206144 will handle request.
Accepted connection from 127.0.0.1 on port 6858
Process 22099, thread 139765561169664 will handle request.
Accepted connection from 127.0.0.1 on port 7370
Process 22099, thread 139765544384256 will handle request.
Accepted connection from 127.0.0.1 on port 7882
Accepted connection from 127.0.0.1 on port 8394
Process 22099, thread 139765502420736 will handle request.
Process 22099, thread 139765544384256 will handle request.
Accepted connection from 127.0.0.1 on port 8906
Accepted connection from 127.0.0.1 on port 9418
Process 22099, thread 139765569562368 will handle request.
Accepted connection from 127.0.0.1 on port 9930
Process 22099, thread 139765535991552 will handle request.
Process 22099, thread 139765569562368 will handle request.
Accepted connection from 127.0.0.1 on port 10442
Process 22099, thread 139765510813440 will handle request.
Accepted connection from 127.0.0.1 on port 10954
Accepted connection from 127.0.0.1 on port 11466
Process 22099, thread 139765519206144 will handle request.
Process 22099, thread 139765527598848 will handle request.
Accepted connection from 127.0.0.1 on port 11978
Process 22099, thread 139765561169664 will handle request.
Accepted connection from 127.0.0.1 on port 12490
Process 22099, thread 139765502420736 will handle request.
Accepted connection from 127.0.0.1 on port 13002
Accepted connection from 127.0.0.1 on port 13514
Process 22099, thread 139765494028032 will handle request.
Process 22099, thread 139765544384256 will handle request.
Accepted connection from 127.0.0.1 on port 14026
Accepted connection from 127.0.0.1 on port 14538
Process 22099, thread 139765535991552 will handle request.
Process 22099, thread 139765569562368 will handle request.
Accepted connection from 127.0.0.1 on port 15050
Process 22099, thread 139765552776960 will handle request.
Accepted connection from 127.0.0.1 on port 15562
```

```
 100%     12 (longest request)
 ~/CSC_3150/Assignment-2/s/3/thread_poll  on git main ↑1 ?4
 ab -n 5000 -c 10 http://localhost:8000/
This is ApacheBench, Version 2.3 <$Revision:  Follow link (cmd + click)  .net/
Copyright 1996 Adam Twiss, Zeus Technology L
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:   1.239 seconds
Complete requests:      5000
Failed requests:        0
Total transferred:      23460000 bytes
HTML transferred:       23130000 bytes
Requests per second:    4035.39 [#/sec] (mean)
Time per request:       2.478 [ms] (mean)
Time per request:       0.248 [ms] (mean, across all concurrent requests)
Transfer rate:          18490.29 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.7      0       6
Processing:     0    2   1.7      2      12
Waiting:        0    2   1.6      1      12
Total:          0    2   1.7      2      12

Percentage of the requests served within a certain time (ms)
  50%      2
  66%      3
  75%      3
  80%      4
  90%      5
  95%      6
  98%      7
  99%      8
 100%     12 (longest request)
 ~/CSC_3150/Assignment-2/s/3/thread_poll  on git main ↑1 ?4
```

Using 10 threads , accepting 5000 requests