CSC3150 Assignment 2 Report

CHEN Ang (118010009)

Program Design

This program (named *Froggy*) is a stripped-down implementation of the classic *Frogger* game. The user needs to control Froggy the frog (green `@`) using `[w][A][S][D]` keys to cross the `(ROW-2) x COL` river. Froggy is afraid of water `(' ')`, so it has to step on the floating logs (`=====`) of varied length and speed to get to the opposite bank. The program saves the map in a `ROW x (RIVER_SIDE_LEN + COL + RIVER_SIDE_LEN)` 2D array. The `RIVER_SIDE_LEN` is used for spawning logs on the the sides of the screen, which would remain invisible to the gamer. The program also utilizes multithreading, in particular POSIX Threads (pthreads), to simulate the individual behavior of Froggy and the floating logs. Specifically, three types of threads are used to control the game:

1. `pth_kb` (keyboard thread) executes `check_kb(void* _)`, which controls the frog movement by fetching keyboard strokes continuously from the gamer. Upon receiving a keyboard hit, it decodes the stroke and either updates the frog position and redraw the frog (`[w][A][S][D]/[w][a][s][d]`) or alters the control flow of the game (`[Q]/[q]` for quitting the game), or, for bonus, controls the speed of the logs if it receives keys `[<]/[>]`. Beside fetching keyboard strokes, this thread is also responsible for checking the status of Froggy continuously and exits once the game has to end. We let the main thread `pthread_join` the keyboard thread. This way the keyboard thread serves as a signal of the end of the game, telling us when to `pthread_cancel` all other threads.
2. `pth_slog[1:ROW-2]` (log-spawn threads) are `ROW-2` threads executing `spawn_logs_on_row(void* y)`, each responsible for indefinitely spawning logs on the side of row `*(int*)y` of the map. The spawning is random both in the log length and the interval between successive logs. The random length is generated by the help of `rand()` in `stdlib` and the interval between logs is randomized by `msleep`ing a random amount of time.
3. `pth_dlog[1:ROW-2]` (log-drift threads) are `ROW-2` threads executing `drift_logs_on_row(void* y)`, each responsible for indefinitely drifting logs on row `*(int*)y`, and printing the updated row on the screen. The direction of the drifting is altered on each row, and the speed of the drifting is randomly selected in an interval decided by `LOG_SLEEP_US`.

It is important to correctly place mutex locks in a multi-threading program so as to avoid resource conflict among threads. In this program a mutex is locked whenever a `pth_dlog` is drawing the updated row onto the screen, and is unlocked once the drawing is finished. The resource in conflict in this case is the `stdout`. Since drawing of each row is realized via multiple single-character `printf`-ing instead of a single `printf`, without the mutex different `printf` commands from different rows would likely be intertwined, producing glitchy graphics.

Problems I met and How I Solved Them

The biggest problem I met was with synchronization of the threads. I did not know specifically where in the code is the problem when I saw the glitchy output at first, but I had a vague feeling it was due to some missing mutex. After a bit digging and experimenting (e.g. printing variables on the screen) I found out it was in due to the multi-thread conflict of `printf` in `drift_logs_on_row`, and was able to fix it by putting a mutex around that part of code. Another problem I met was also related to synchronization. When the `pth_kb` thread exits from the game, how can I tell all the threads that are controlling logs, most likely at sleep, to exit as well? At first I tried to use `pthread_cond_timedwait()` in the log threads and signal them to wake up from sleep using `pthread_cond_broadcast()` once `pth_kb` has exited. But for some reason the output remains glitchy and I couldn't figure out why. After some life struggling I eventually realized I could just `pthread_cancel` all the log threads without worrying memory leakage or anything because these threads never actually allocate any memory. I could simply pass the `y` arguments I declare in `main` to them as automatic variables and never worry about it anymore. Third problem I met was the VM ran so slow and laggy on my laptop, and so I switched to WSL2.

Environment

The program is run on `WSL2 (Windows Subsystem for Linux)`, `Ubuntu 20.04.1 LTS (x64)`, kernel `4.19.128-microsoft-standard`.

How to Run the Programs

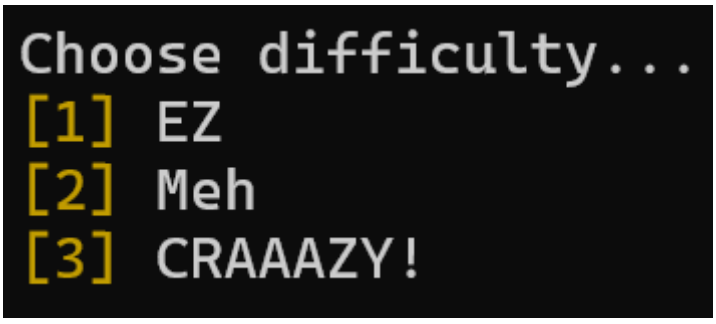
In `source` directory, type into the console

```
make && ./a.out
```

to compile and execute the program.

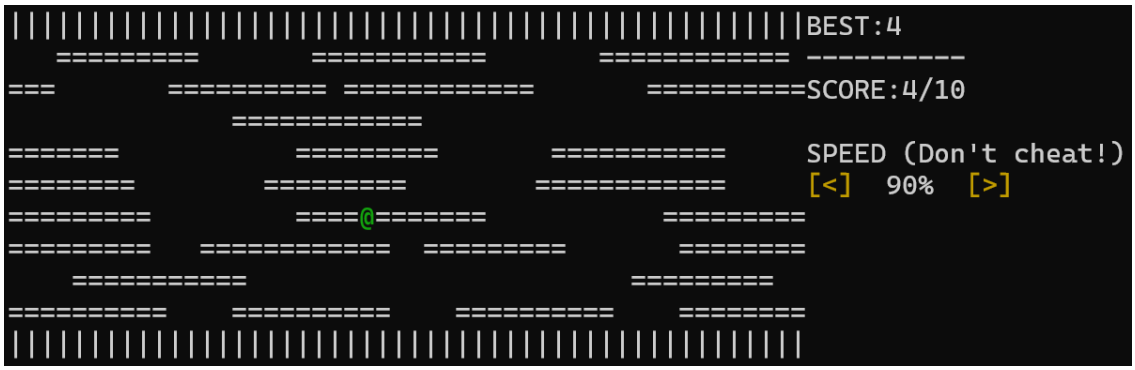
Screenshots

- Prompt of choosing difficulty level.

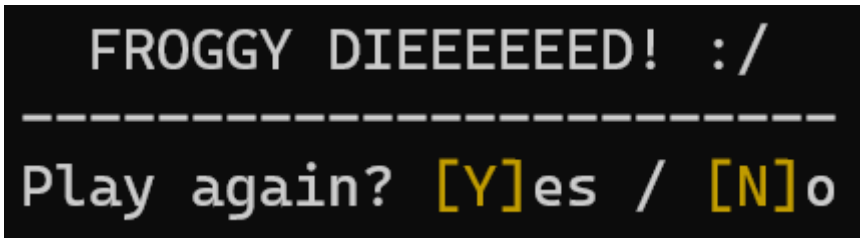


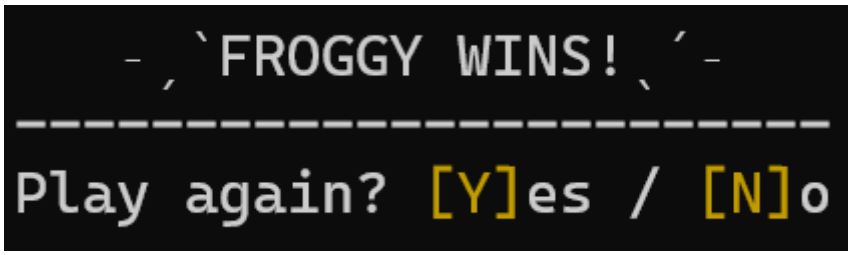
- Game countdown, tips, score board, and speed control panel.

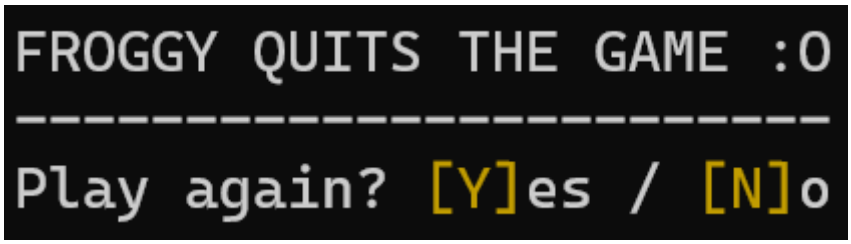
- In-game screenshot. Log speed altered to `90%`.



- Game prompts when Froggy died, wins the game, and when the user quits the game.







What I Learned

I learned the basics of how to create, exit, and cancel pthreads. I also learned the importance of mutexes in multi-threading programs for avoiding resource conflicts among threads, and how to initialize and destroy them in C code. Finally, I learned how to use escape sequences to do powerful things in a terminal.

BONUS

Program Design

I did not use any GUI library but instead chose to implement a TUI, as can be seen on the screenshots.

Both the length of the logs and the interval between successive logs are randomized via `rand()` and `usleep(rand())`.

The slide bar is now a control panel showing the speed percentage which can be controlled using `[<]` and `[>]` keys.